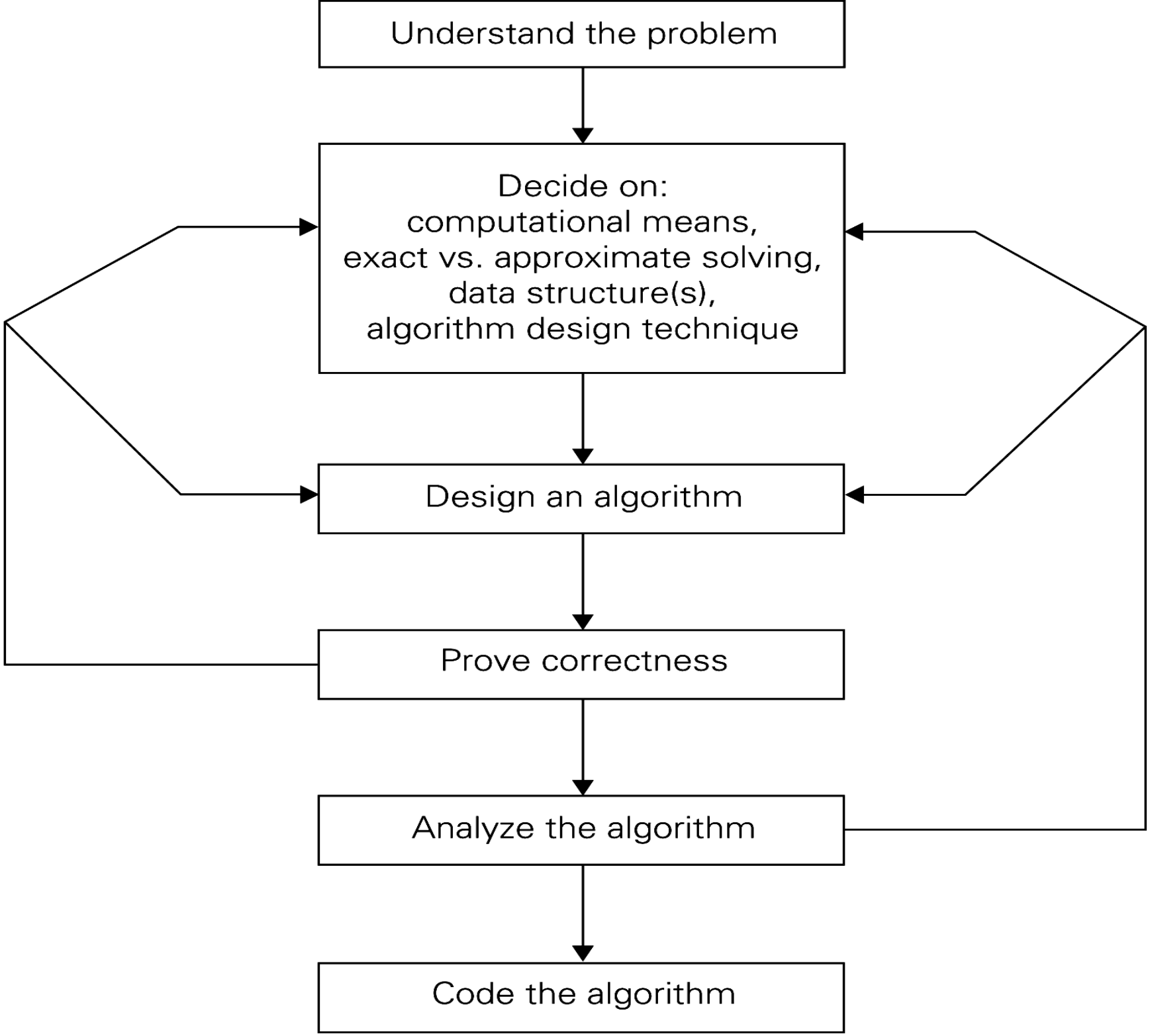
JICSCI803
Algorithms and Data Structures
March to June 2019

Highlights of Lecture 06

B-Trees and Treaps

Algorithm Design and Analysis Process



Algorithm Analysis Framework

Measuring an input's size
Measuring running time
Orders of growth (of the algorithm's efficiency function)
Worst-base, best-case and average-case efficiency

Refresh

Binary Tree
Binary Tree Traversal
Binary Tree Search
AVL Binary Tree

AVL Trees

- An AVL (Adelson-Velski and Landis) tree is a binary search tree with a balance condition.
- Every node must have left and right subtrees which differ in height by at most 1”

B-Tree

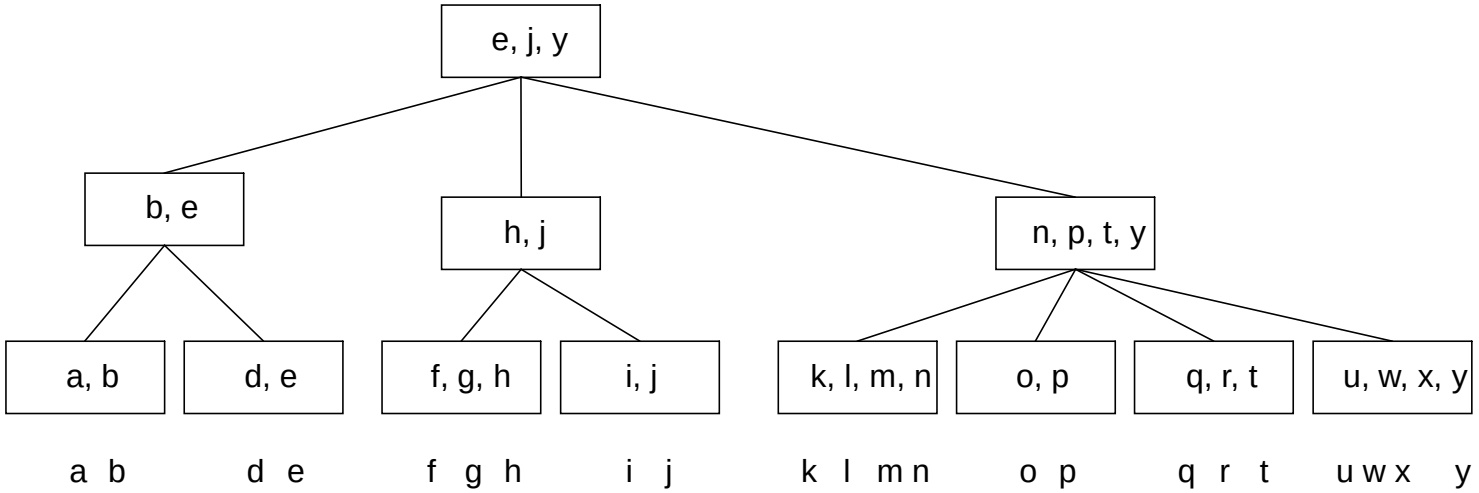
In computer science, a B-tree is a self-balancing tree data structure that **keeps data sorted** and allows searches, sequential access, insertions, and deletions in logarithmic time. The B-tree is a **generalization of a binary search tree** in that **a node can have more than two children** (Comer 1979, p. 123). Unlike self-balancing binary search trees, the B-tree is optimized for systems that read and write large blocks of data. B-trees are a good example of a data structure for external memory. It is commonly used in databases and filesystems.

2-4 Trees (The easy version)

- A 2-4 tree is a tree with the following properties:

1. Every node (except possibly the root) has between two and four nodes.
2. All leaves of the tree are at the same depth.
3. Items are stored in the leaves of the tree. Each internal node contains, for each child, the value of the largest key in the sub tree rooted at that child.
4. The items in the leaves appear in order from left to right.

2-4 Trees-- An example



2-4 Trees: Searching

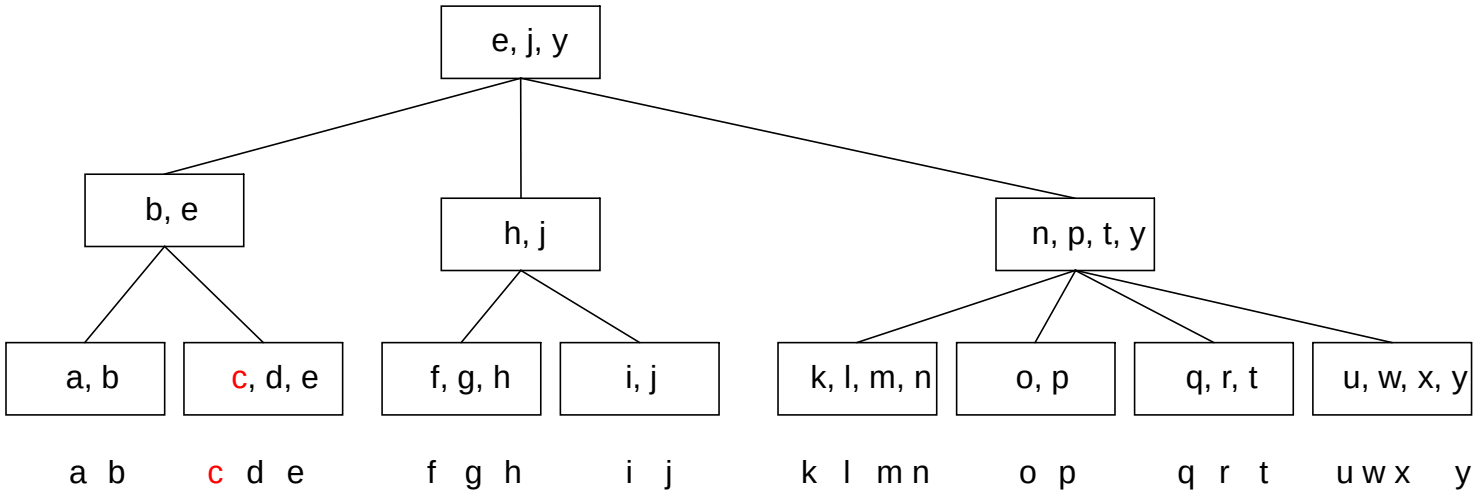
– Similar to searching a binary tree
– At each level go to the leftmost child with value ≥ key

2-4 Trees: Insertion

- Find where the item is to be inserted
- Insert the item
- Update the node
  1. Insertion into a 2-node $\Rightarrow$ 3-node
  2. Insertion into a 3-node $\Rightarrow$ 4-node
  3. Insertion into a 4-node $\Rightarrow$5-node
     - Split this node and update its parent.
       1. Repeat with the parent if necessary.
       2. Create a new root layer if necessary.

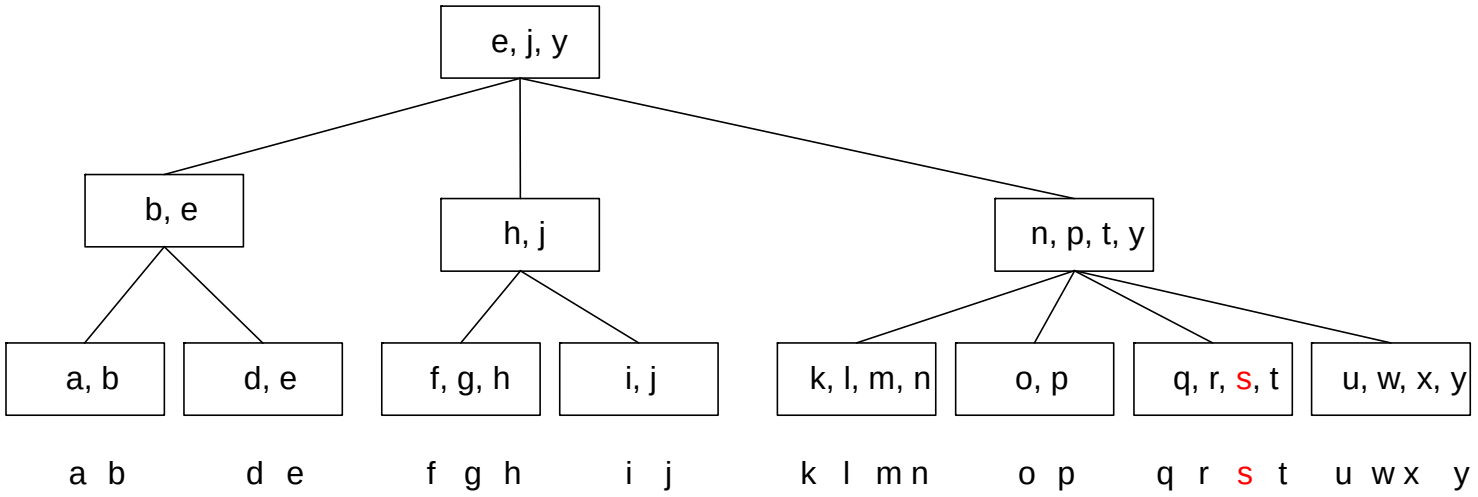
2-4 Trees

– E.g. insert “c”: 2-node becomes 3-node



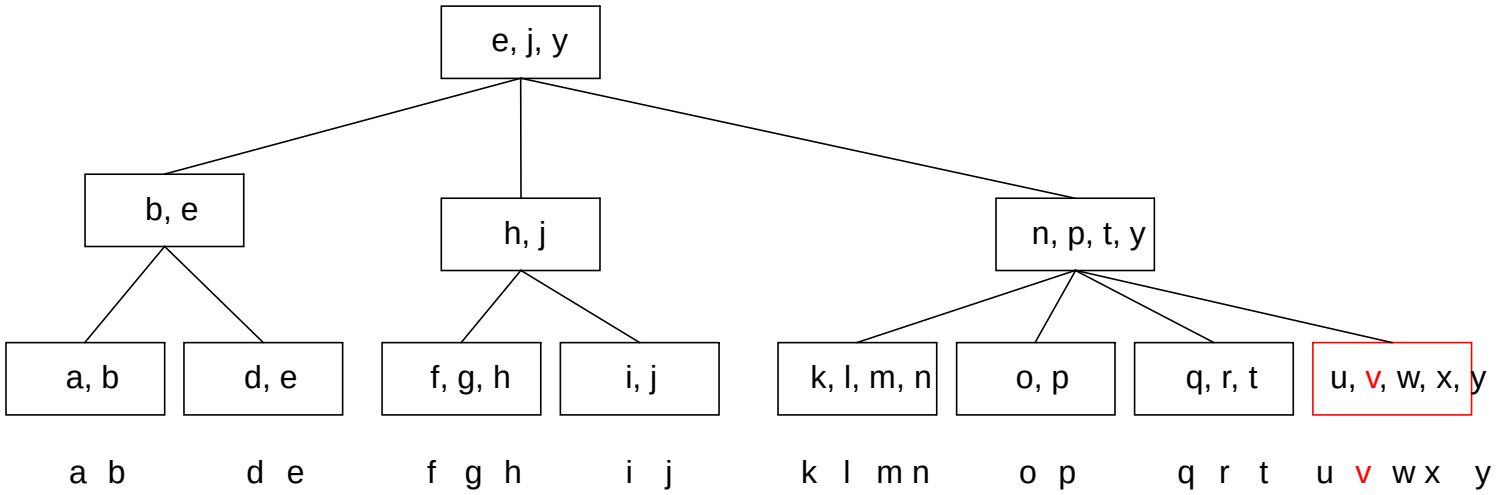
2-4 Trees

–E.g. insert “s”: 3-node becomes 4-node



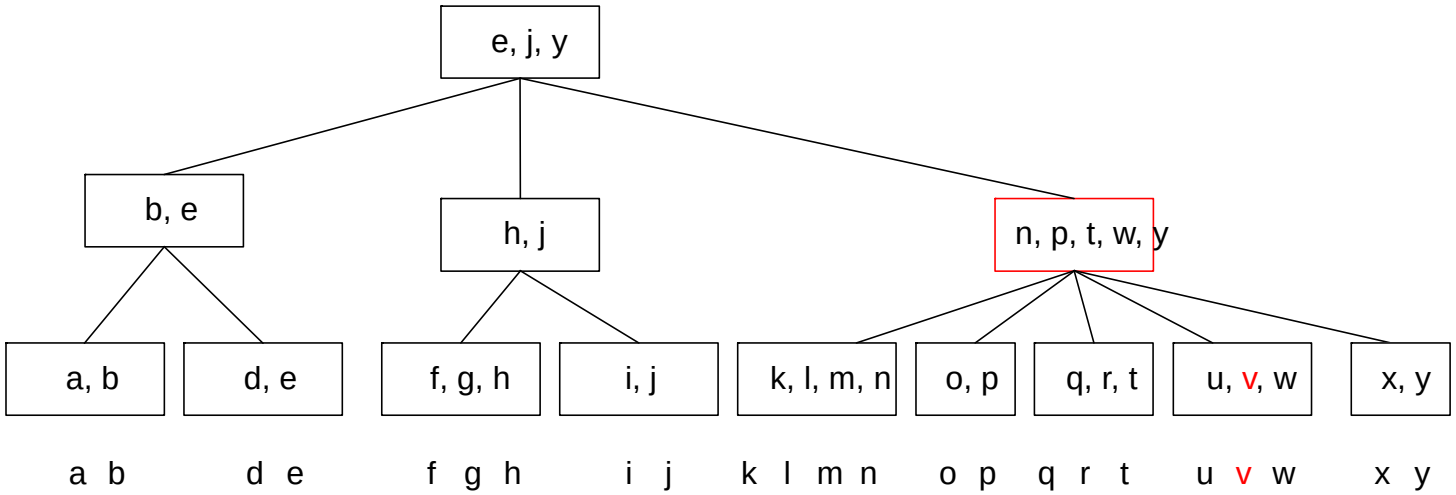
2-4 Trees

–E.g. insert “v”: 4-node becomes 5-node



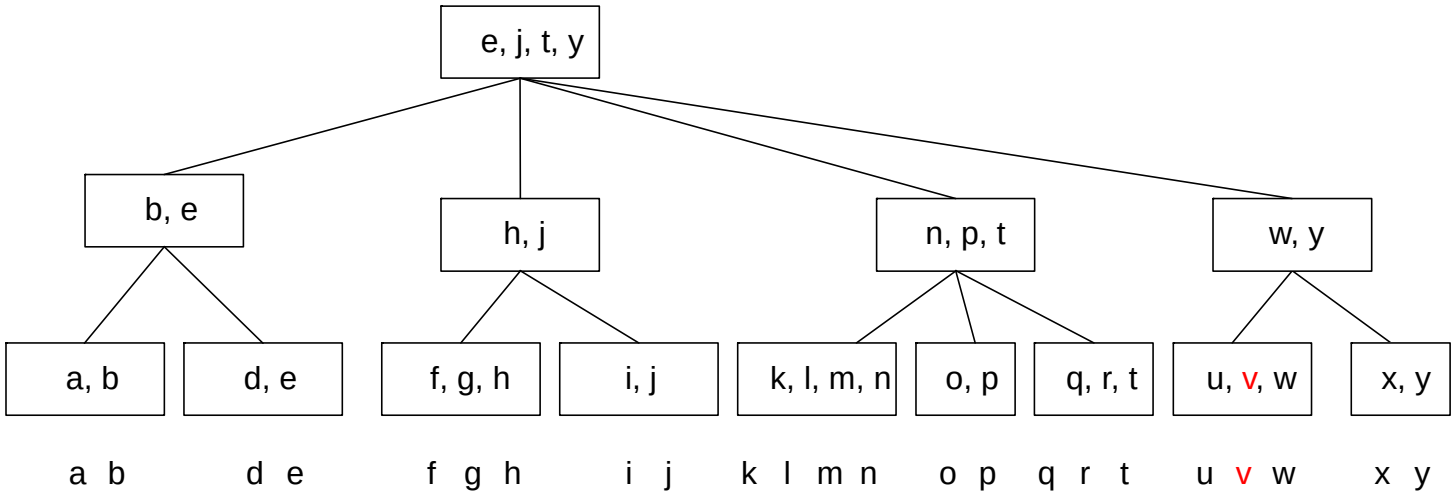
2-4 Trees

–E.g. insert “v”: split the 5-node



2-4 Trees

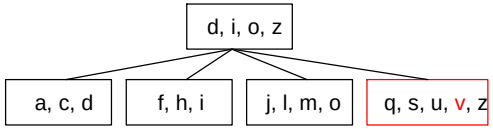
– E.g. insert “v”: split this 5-node

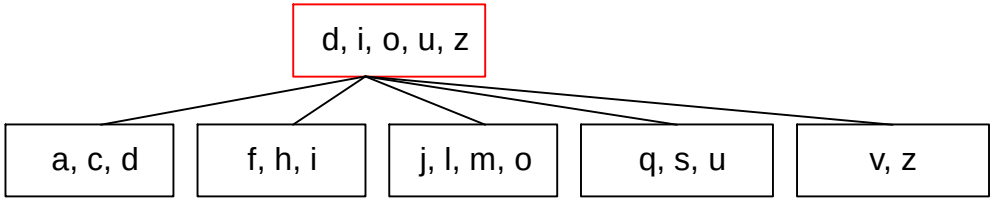


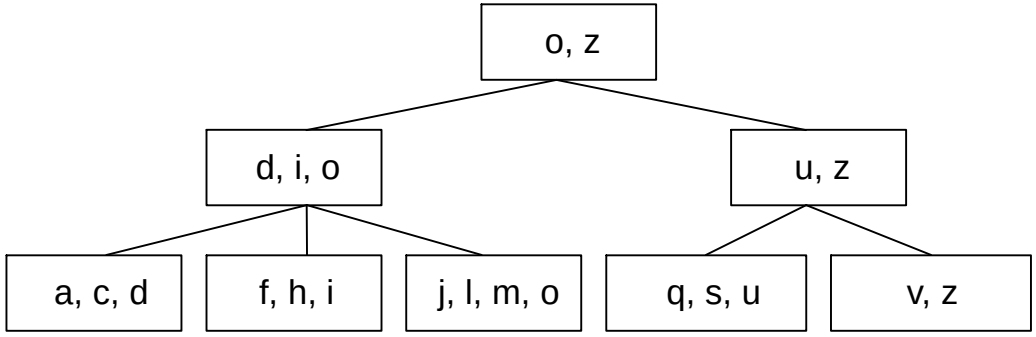
2-4 Trees

- If the root node becomes a 5-node, split the root node and create a new root
- This increases the height of the tree by one.

2-4 Trees


d, i, o, z
a, c, d
f, h, i
j, l, m, o
q, s, u, v, z


d, i, o, u, z
a, c, d
f, h, i
j, l, m, o
q, s, u
v, z


o, z
d, i, o
u, z
a, c, d
f, h, i
j, l, m, o
q, s, u
v, z

2-4 Trees: Deletion

– Find where the item to be deleted is located

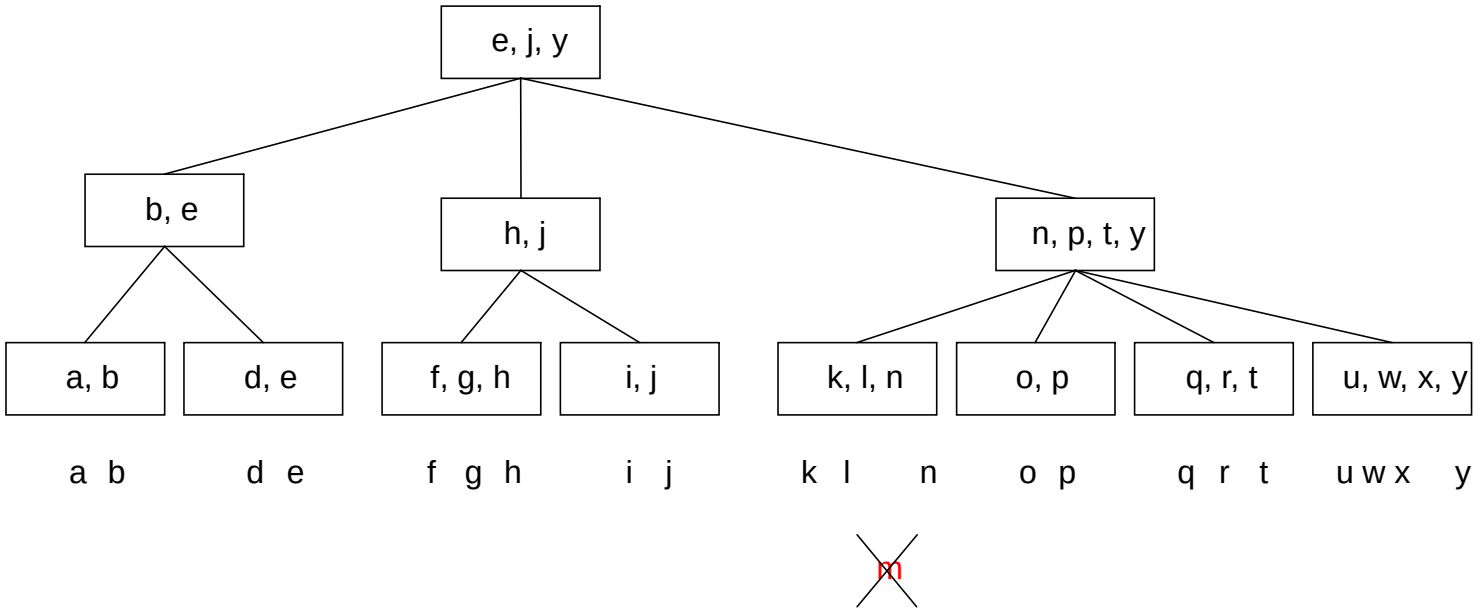
– Delete the item

– Update the node

1. Deletion from a 4-node ⇒3-node
2. Deletion from a 3-node ⇒2-node
3. Deletion from a 2-node ⇒1-node
   - If the 1-node has a sibling with more than two children redistribute the children
   - If not, collapse the 1-node and one of its siblings into a 3-node and update the parent
   - This may cause the parent to become a 1-node
   - Fix this recursively
   - If the root becomes a 1-node, remove it

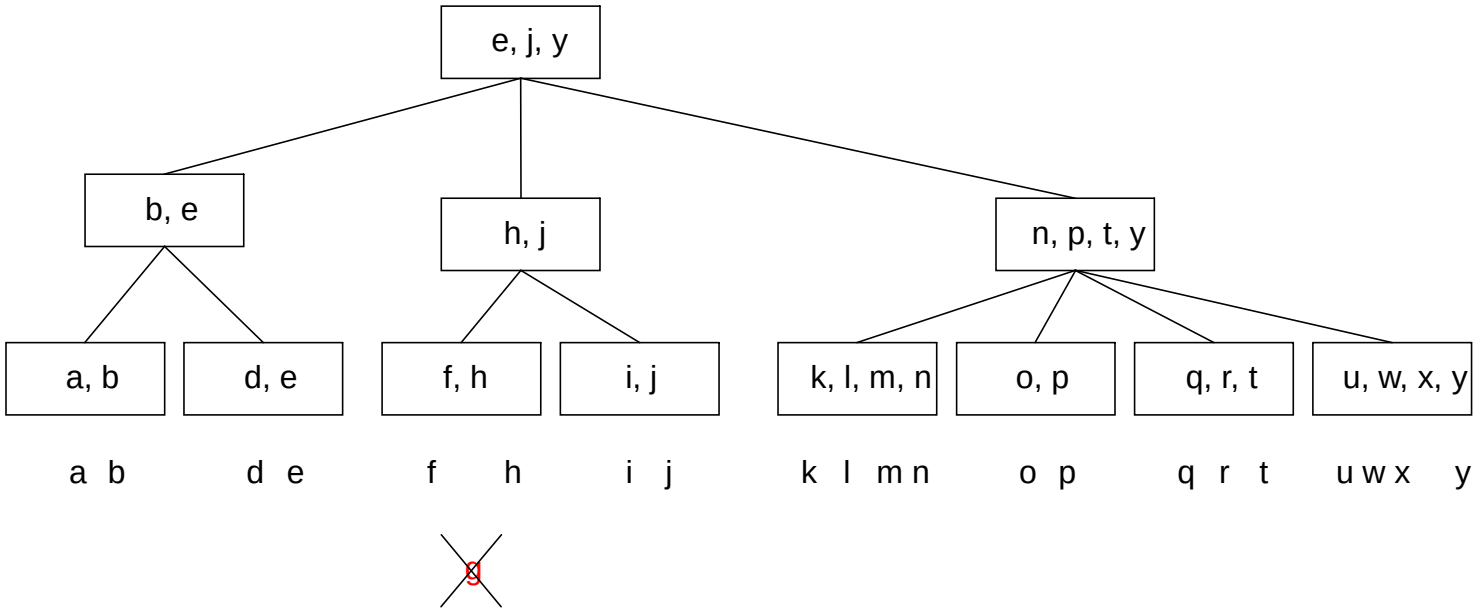
2-4 Trees

–E.g. delete “m”: 4-node becomes 3-node



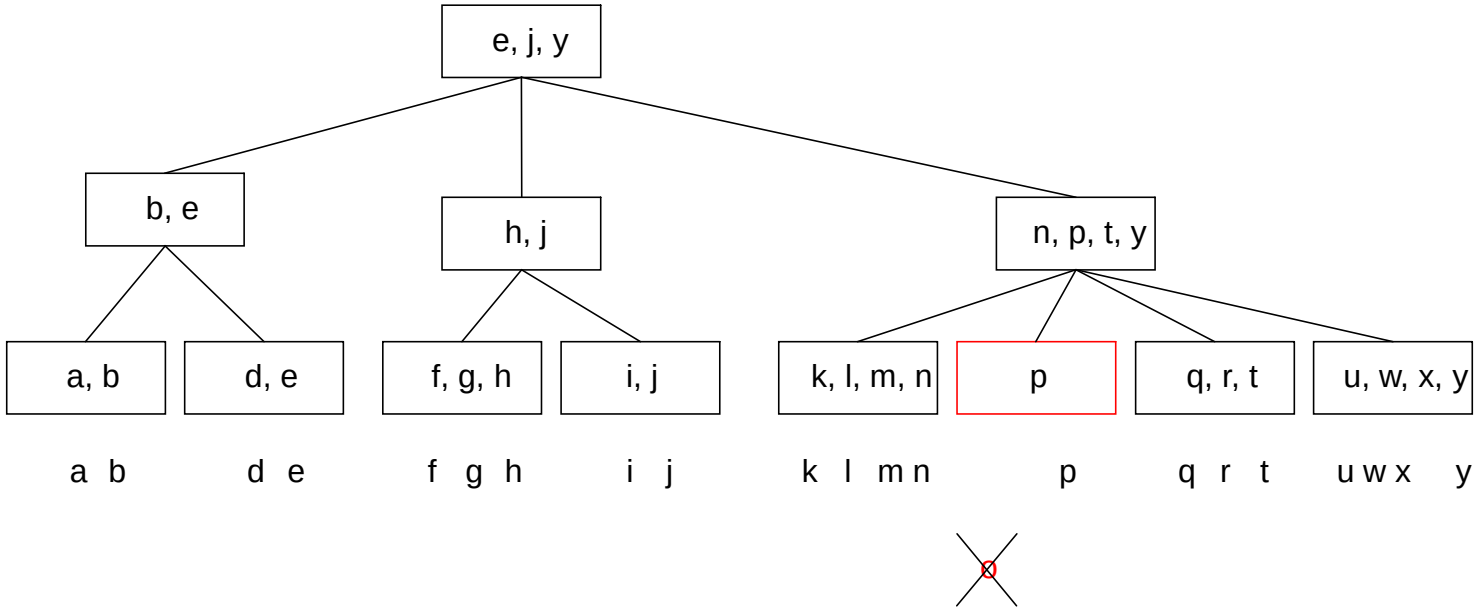
2-4 Trees

–E.g. delete “g”: 3-node becomes 2-node

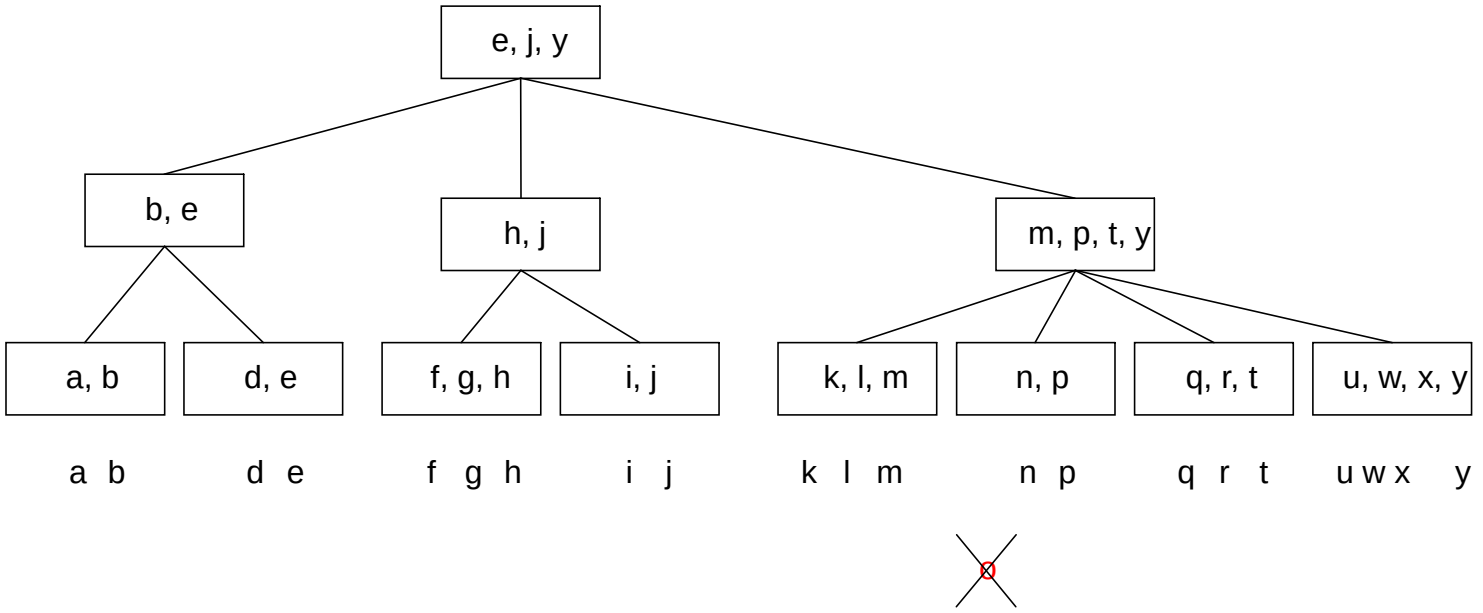


2-4 Trees

– E.g. delete “o”: 2-node becomes 1-node
– share children with a sibling

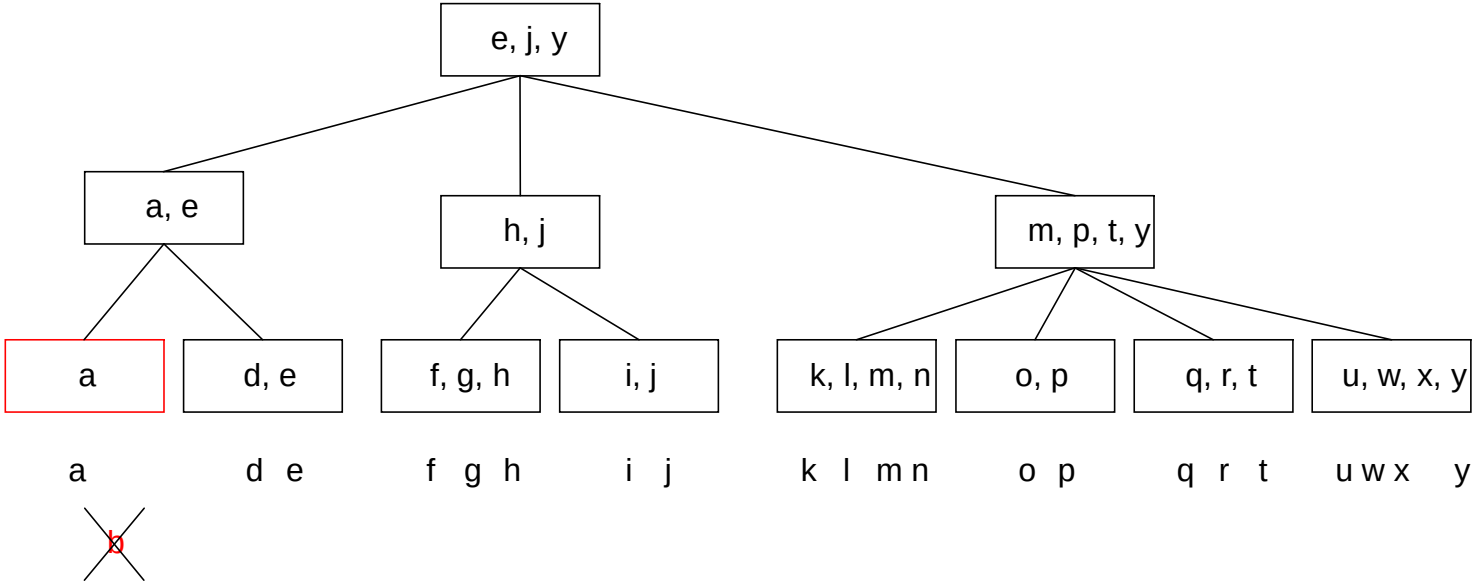


2-4 Trees


e, j, y
b, e
h, j
m, p, t, y
a, b
d, e
f, g, h
i, j
k, l, m
n, p
q, r, t
u, w, x, y
a b
d e
f g h
i j
k l m
n p
q r t
u w x y
∅

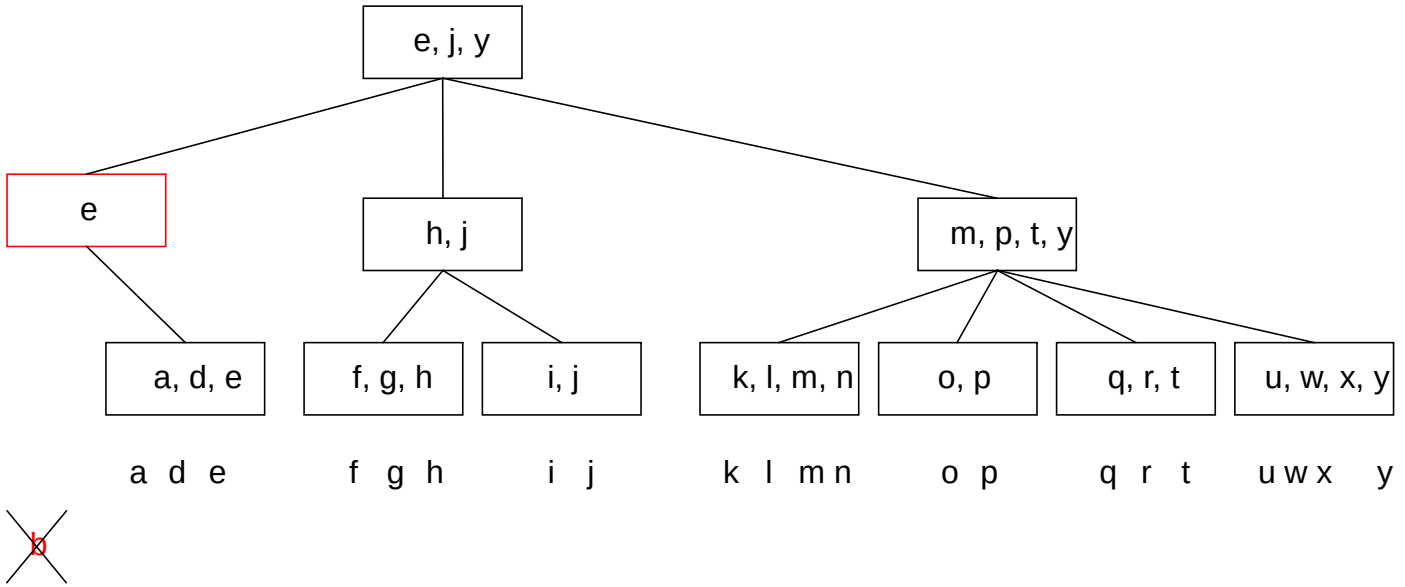
2-4 Trees

–E.g. delete “b”: 2-node becomes 1-node
–merge with sibling

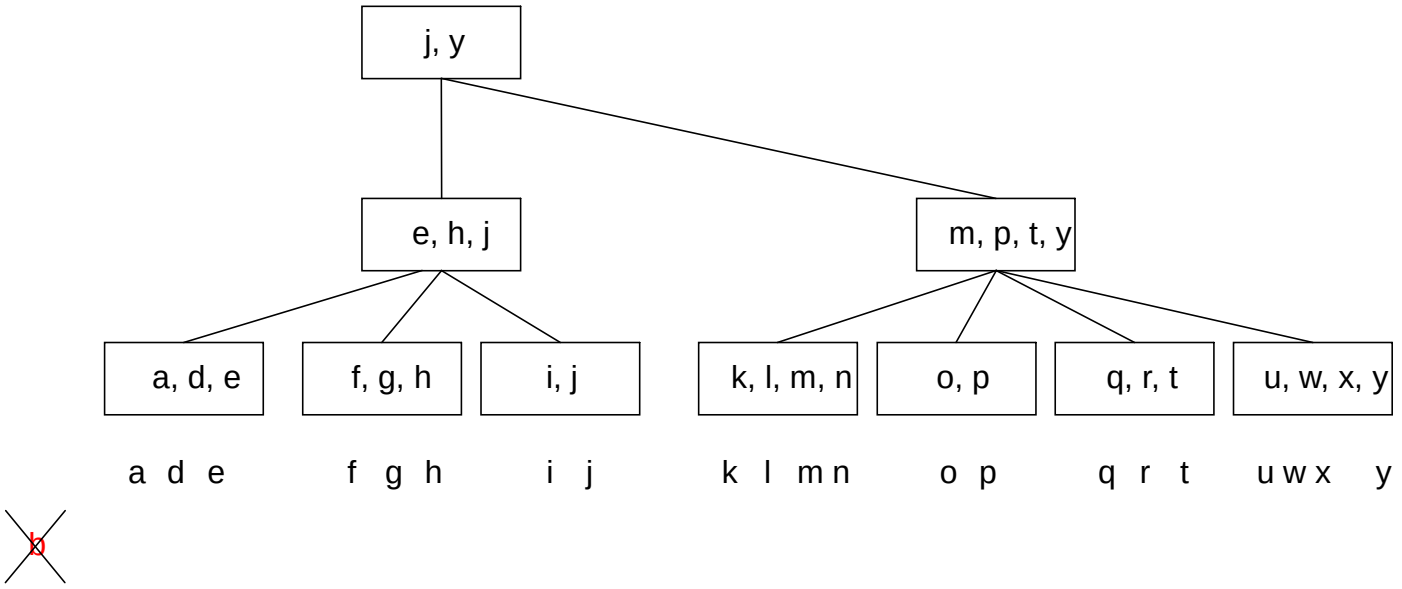


2-4 Trees

–repeat for new 1-node



2-4 Trees


j, y
e, h, j
m, p, t, y
a, d, e
f, g, h
i, j
k, l, m, n
o, p
q, r, t
u, w, x, y
a d e
f g h
i j
k l m n
o p
q r t
u w x y
b

2-4 Trees: Efficiency

- Height of tree is O(log *n*)
- Searching:
  - Each node checked takes O(1)
  - Search takes O(log *n*)
- Insertion:
  - Split takes O(1) at each level
  - At most log *n* splits (1 per level)
  - Insertion takes O(log *n*)
- Deletion
  - Merge takes O(1)
  - At most log *n* merges
  - Deletion takes O(log *n*)

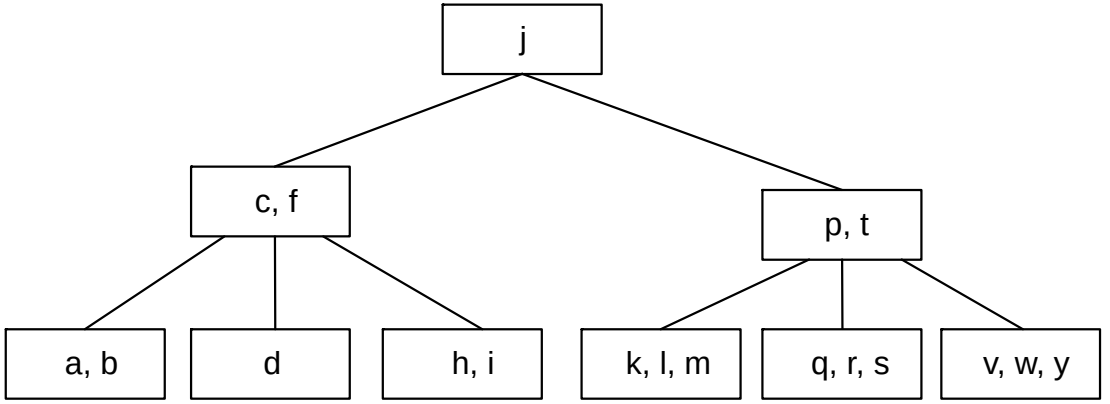
2-4 Trees (The hard version)

– A 2-4 tree is a tree with the following properties:

1. Every internal node (except possibly the root) has between two and four children.
2. The keys within each node are ordered from smallest to largest.
3. Internal nodes have one less key than they have children. Each such key is conceptually positioned between two consecutive children. Its value is larger than the largest key in the subtree to its left and smaller than the smallest key in the subtree to its right.
4. All leaves of the tree are at the same depth.

2-4 Trees

–An example



2-4 Trees: Searching

- Similar to searching a binary tree
- If searching for value, compare value with each key
  - If value == key return the associated data
  - If value < key recursively search the subtree left of key
  - If value > key, repeat the process using the next key, if there is no next key, search the last subtree.

2-4 Trees: Insertion

–Find the leaf where the item is to be inserted

–Insert the item

–Update the node

1. Insertion into a 2-node ⇒3-node
2. Insertion into a 3-node ⇒4-node
3. Insertion into a 4-node ⇒5-node

n-node: has n-1 keys

(n children if not a leaf)

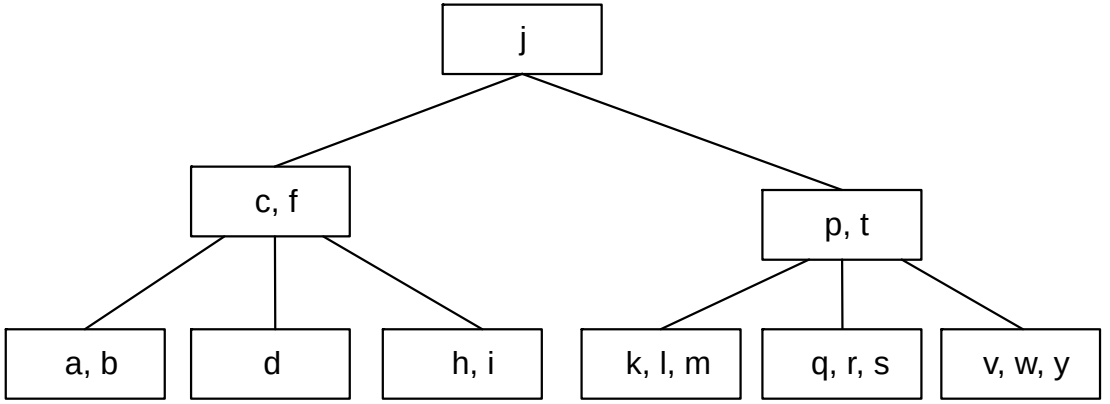
– If an immediately adjacent sibling is not full, send a key from the parent down to this sibling, and a key up from the 5-node to the parent.

– If all such siblings are full, split the node into two by passing the median key up to the parent, and relink subtrees if needed.

1. Repeat with the parent if necessary.
2. Create a new root layer if necessary.

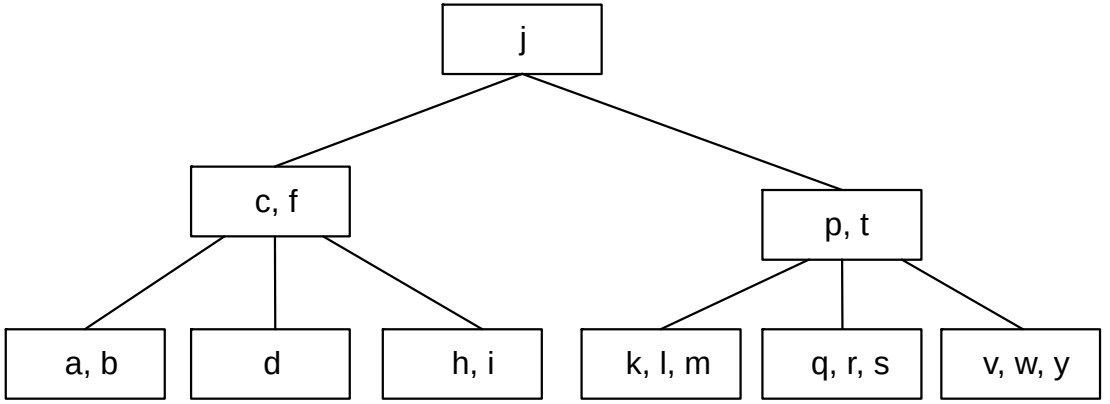
2-4 Trees

Determinate each node is what type of node? i.e. what is the value of x in x-node?



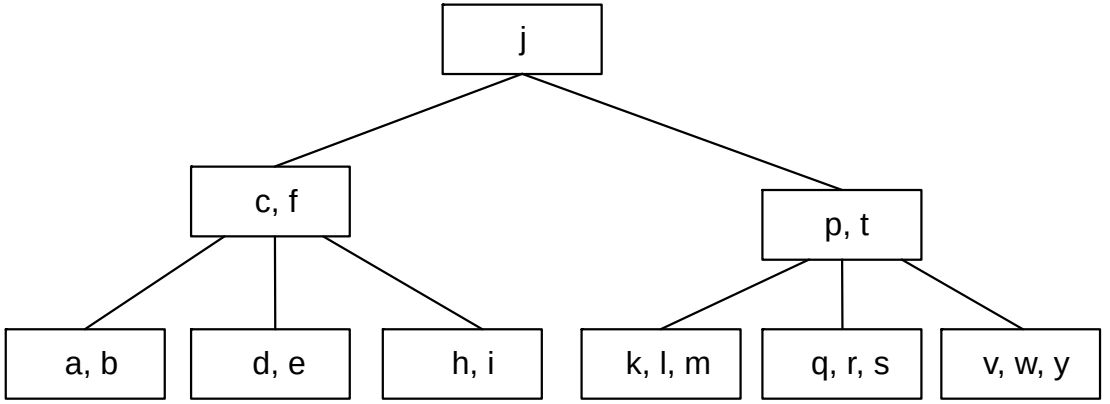
2-4 Trees

– E.g. insert “e”: 2-node becomes 3-node



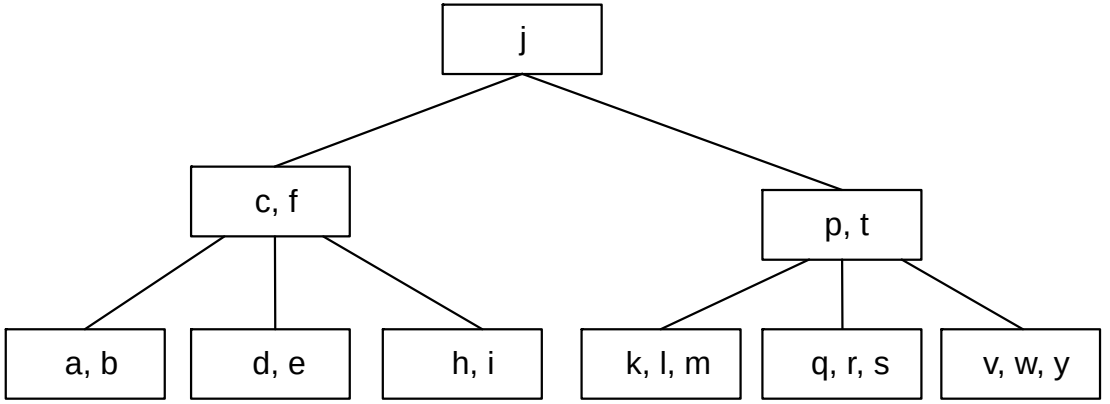
2-4 Trees

– E.g. insert “e”: 2-node becomes 3-node



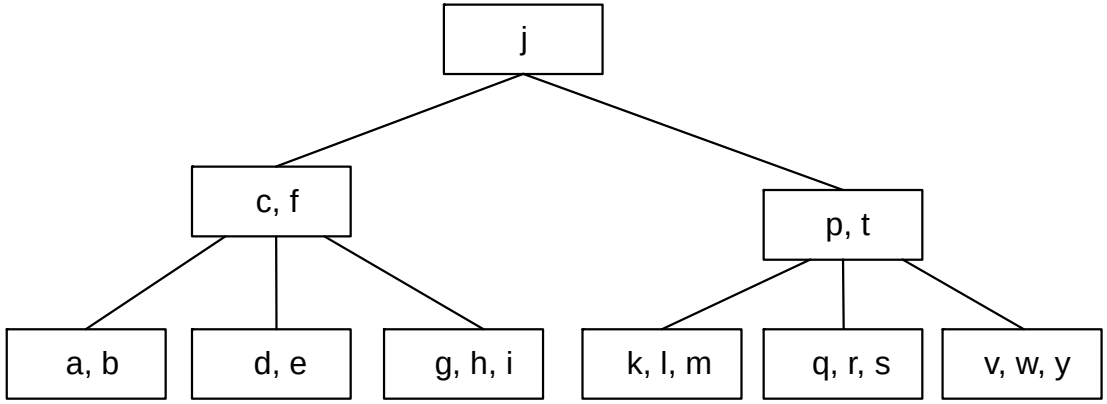
2-4 Trees

– E.g. insert “g”: 3-node becomes 4-node



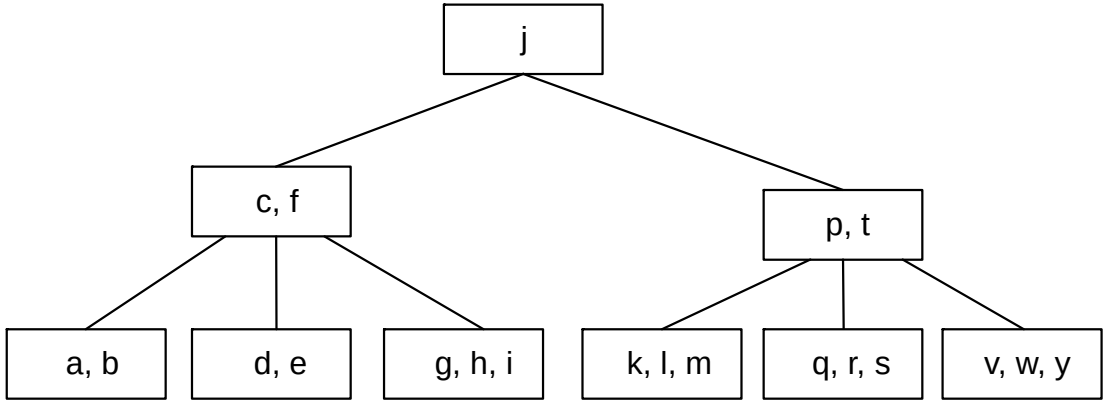
2-4 Trees

– E.g. insert “g”: 3-node becomes 4-node



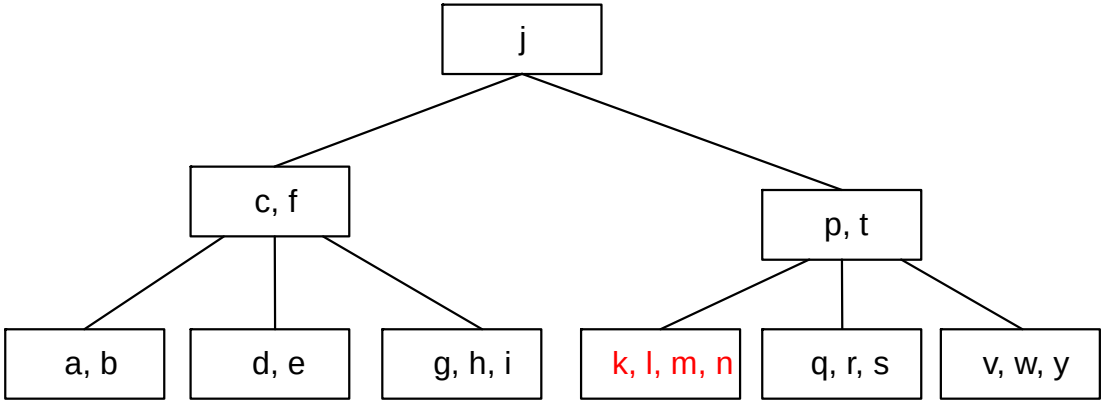
2-4 Trees

– E.g. insert “n”: 4-node becomes 5-node



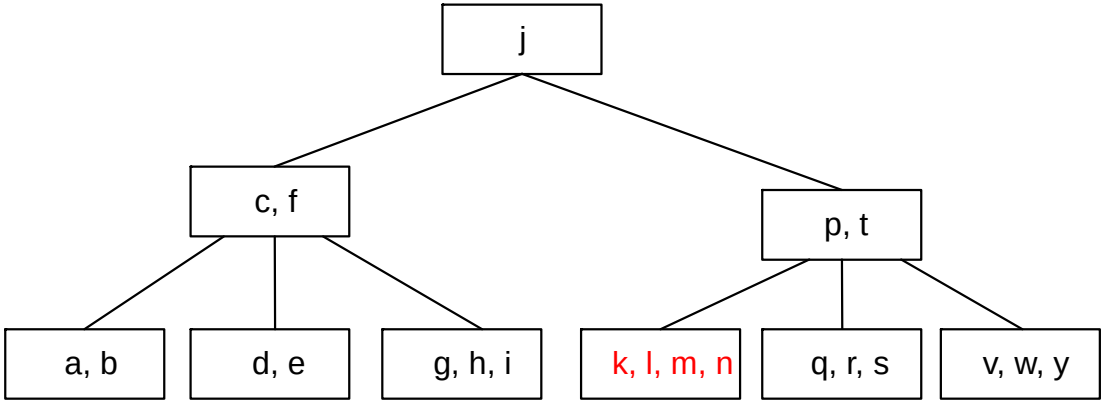
2-4 Trees

– E.g. insert “n”: 4-node becomes 5-node



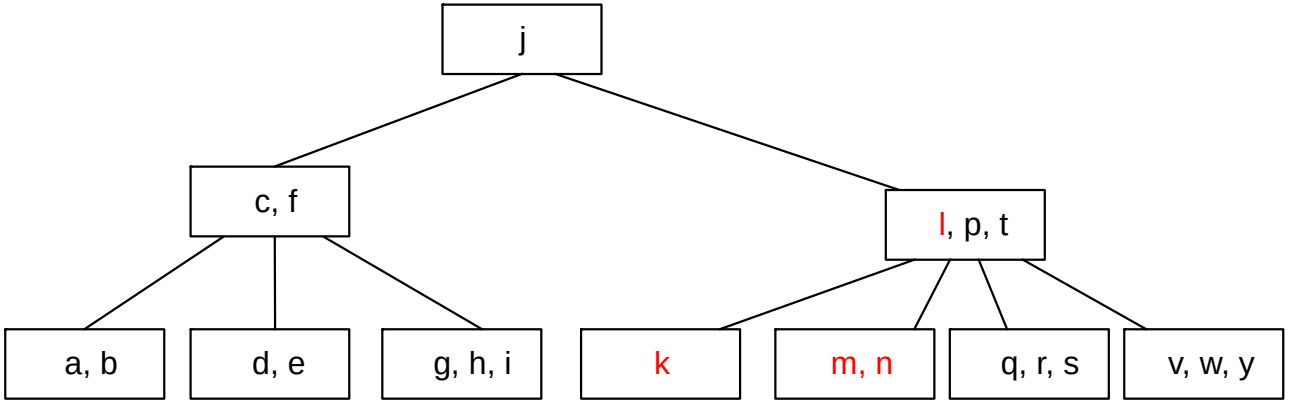
2-4 Trees

– E.g. insert “n”: siblings full, split the 5-node



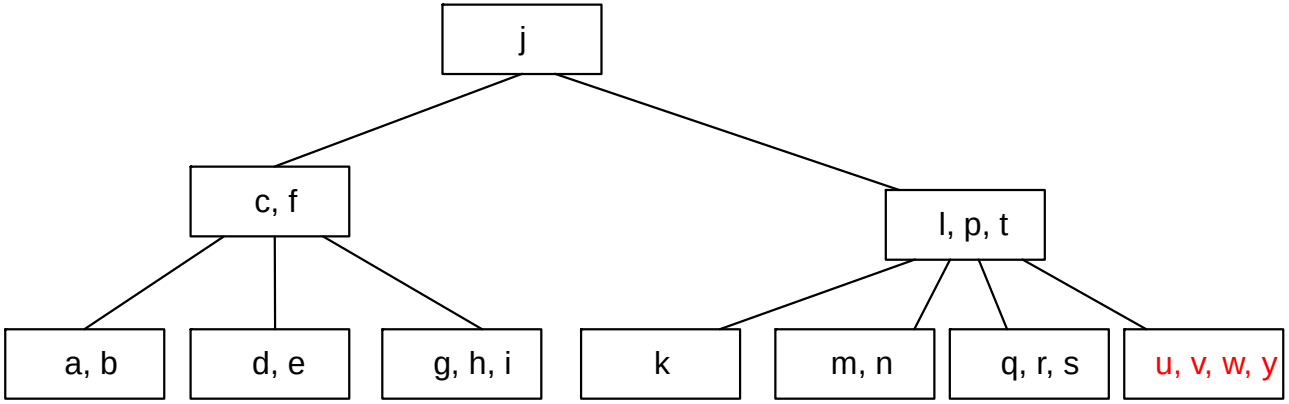
2-4 Trees

– E.g. insert “n”: siblings full, split the 5-node



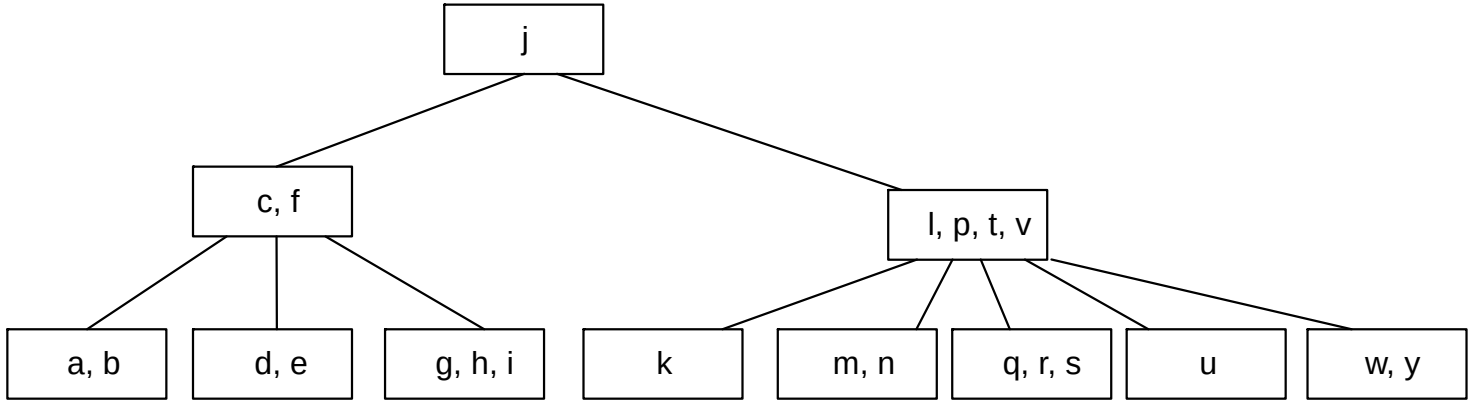
2-4 Trees

– E.g. insert “u”: siblings full, split the 5-node



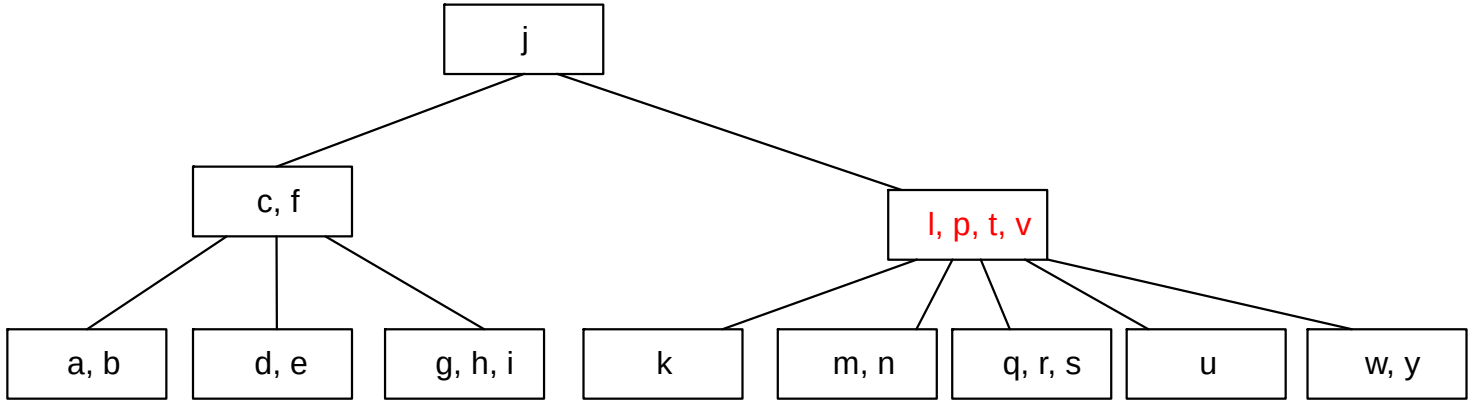
2-4 Trees

– E.g. insert “u”: siblings full, split the 5-node



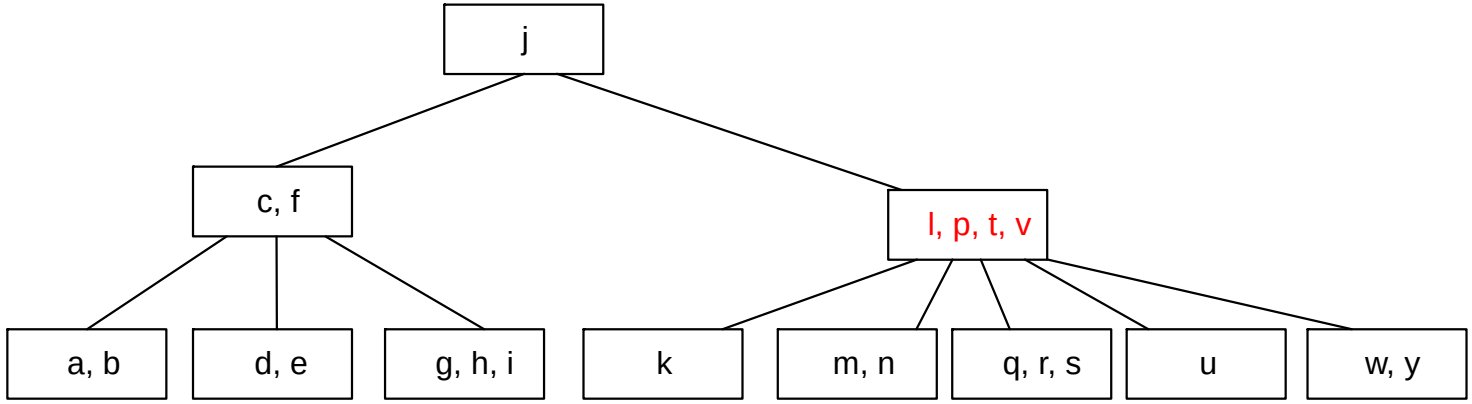
2-4 Trees

– E.g. insert “u”: parent is now a 5-node



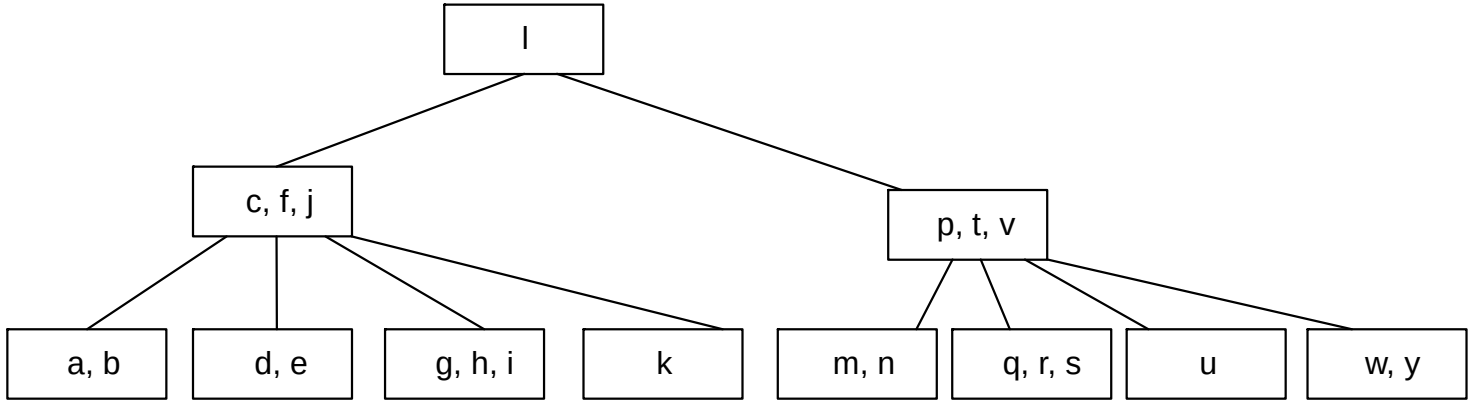
2-4 Trees

– E.g. insert “u”: sibling is not full



2-4 Trees

– E.g. insert “u”: send key from parent down to sibling and key from node up to parent



2-4 Trees

- If the root node becomes a 5-node, split the root node and create a new root containing the median key of the old root
- This increases the height of the tree by one.

2-4 Trees: Deletion

– Find where the item to be deleted is located
– If this is an internal node, swap the item with its immediate inorder successor
– Delete the item
– Update the node

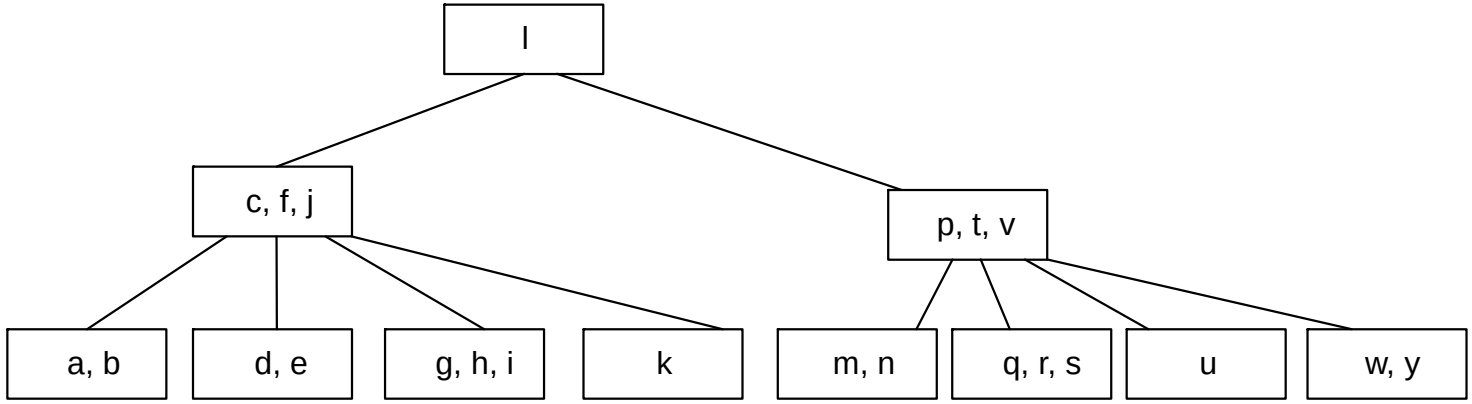
2-4 Trees: Deletion

– Update the node

1. Deletion from a 4-node ⇒ 3-node
2. Deletion from a 3-node ⇒ 2-node
3. Deletion from a 2-node ⇒ 1-node
   - If the 1-node has an immediate sibling with more than one key, send a key from the sibling up to the parent, and a key from the parent down to the node and relink if necessary
   - If not, remove the node and send the key down from the parent into a sibling node
   - This may cause the parent to become a 1-node
   - Fix this recursively
   - If the root becomes a 1-node, remove it

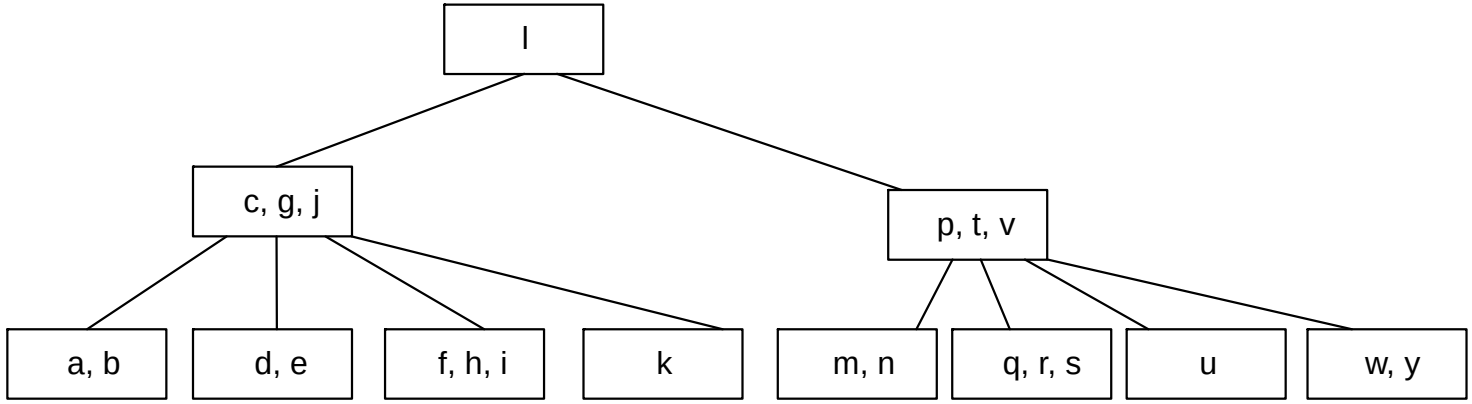
2-4 Trees

– E.g. delete “ f ”: internal node



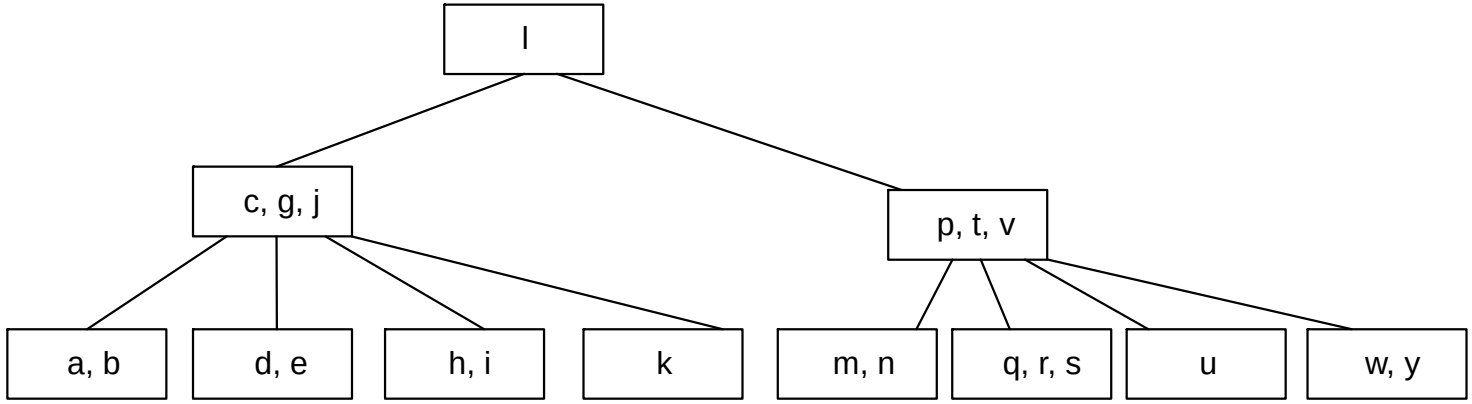
2-4 Trees

– E.g. delete “ f ”: swap with “g”



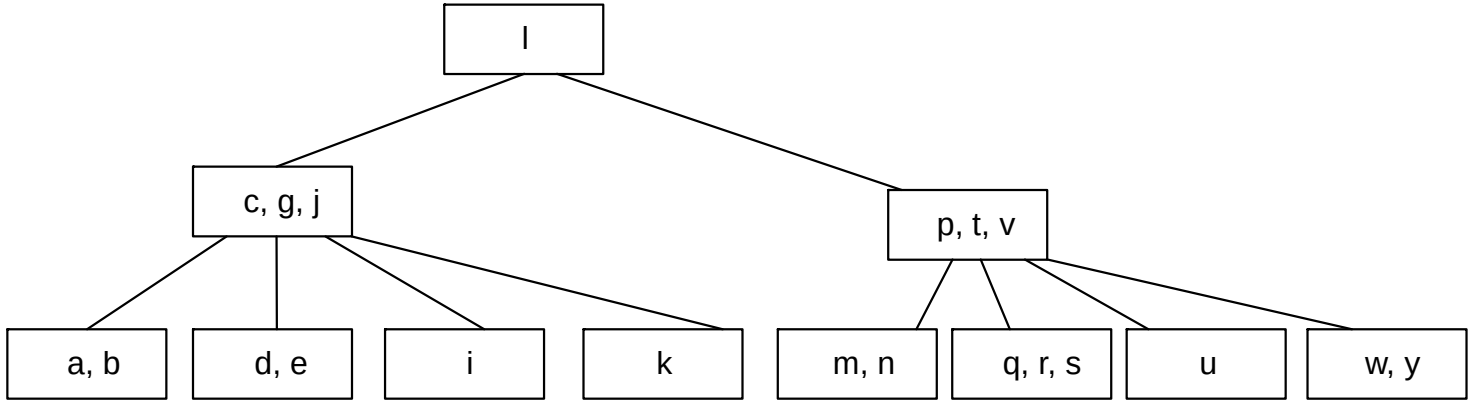
2-4 Trees

– E.g. delete “f”: 4-node becomes 3-node



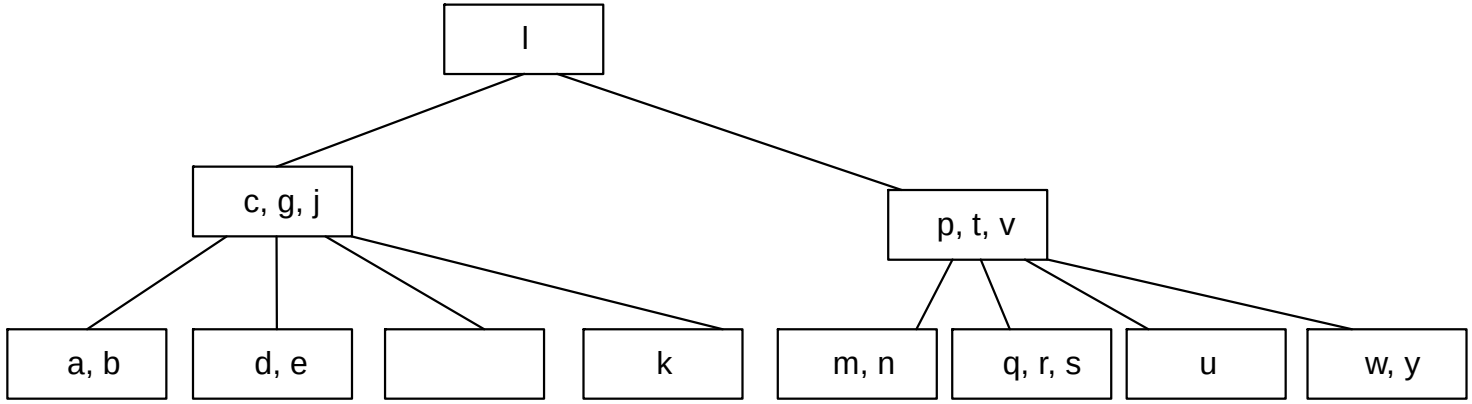
2-4 Trees

– E.g. delete “h”: 3-node becomes 2-node



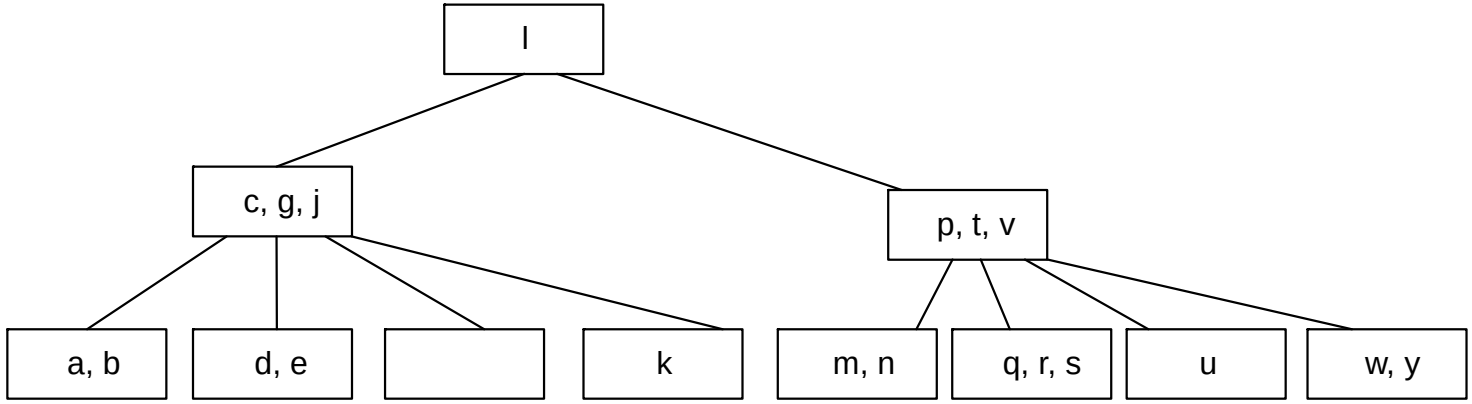
2-4 Trees

– E.g. delete “i”: 2-node becomes 1-node



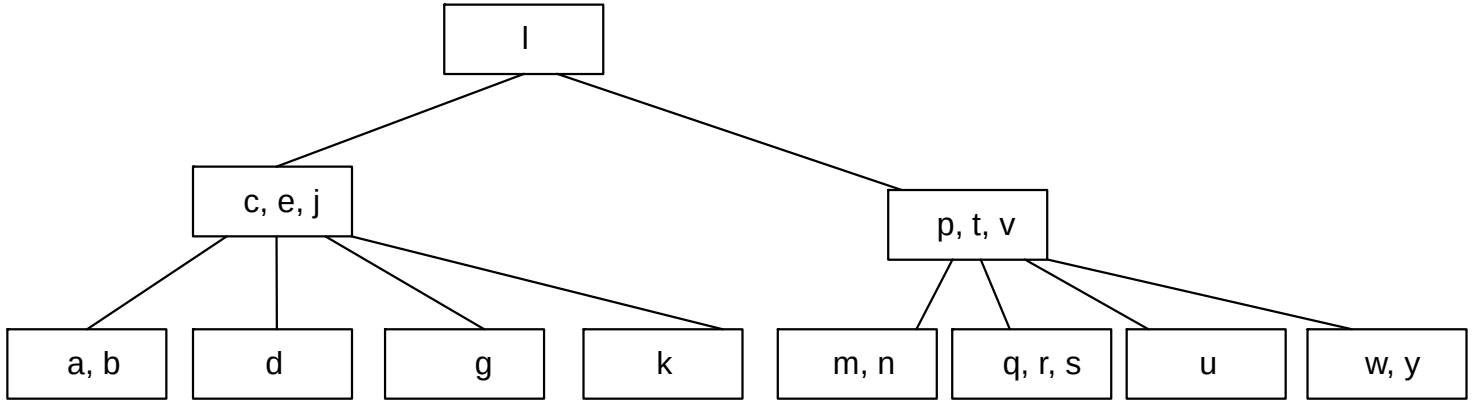
2-4 Trees

– E.g. delete “i”: has a sibling with more than 1 key



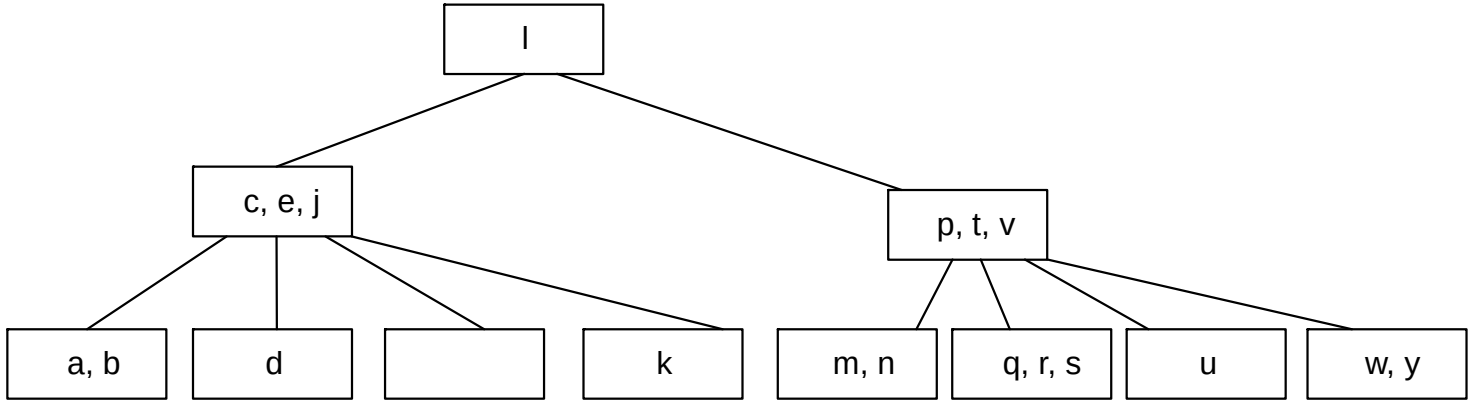
2-4 Trees

– E.g. delete “i”: send a key from sibling to parent, and key from parent to the node



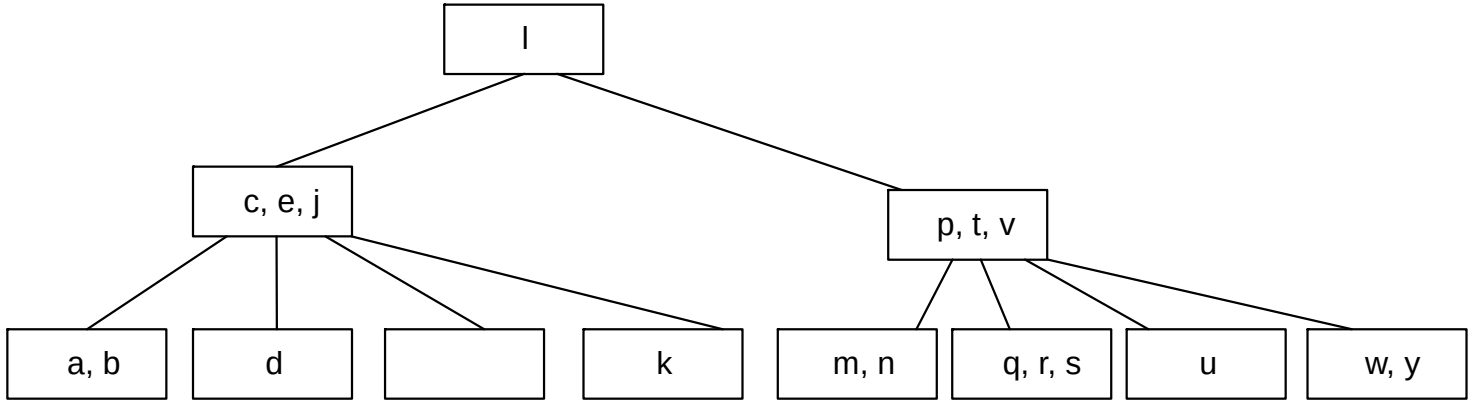
2-4 Trees

– E.g. delete “g”: 2-node becomes 1-node



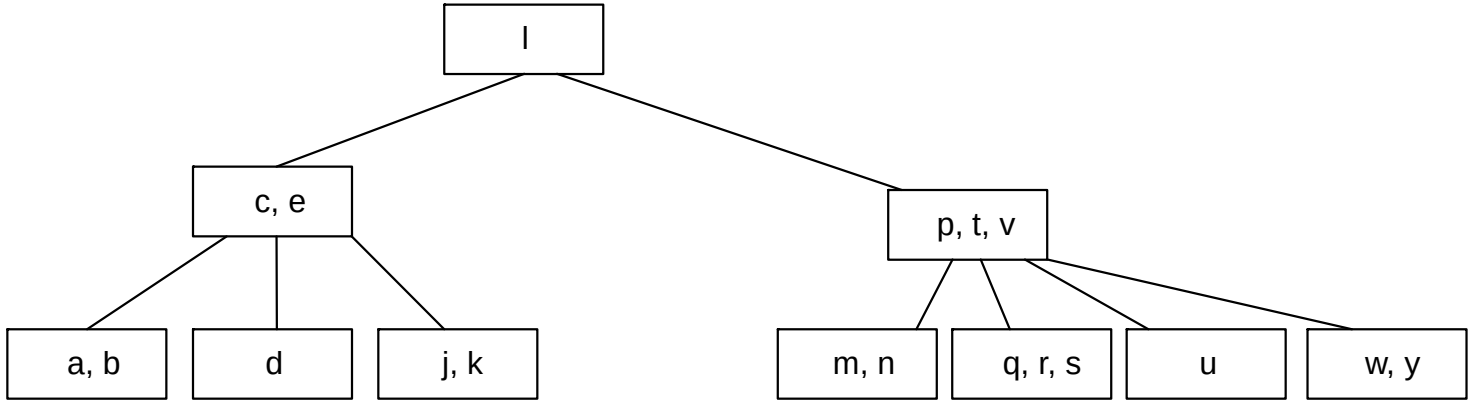
2-4 Trees

– E.g. delete “g”: no siblings with more than one key



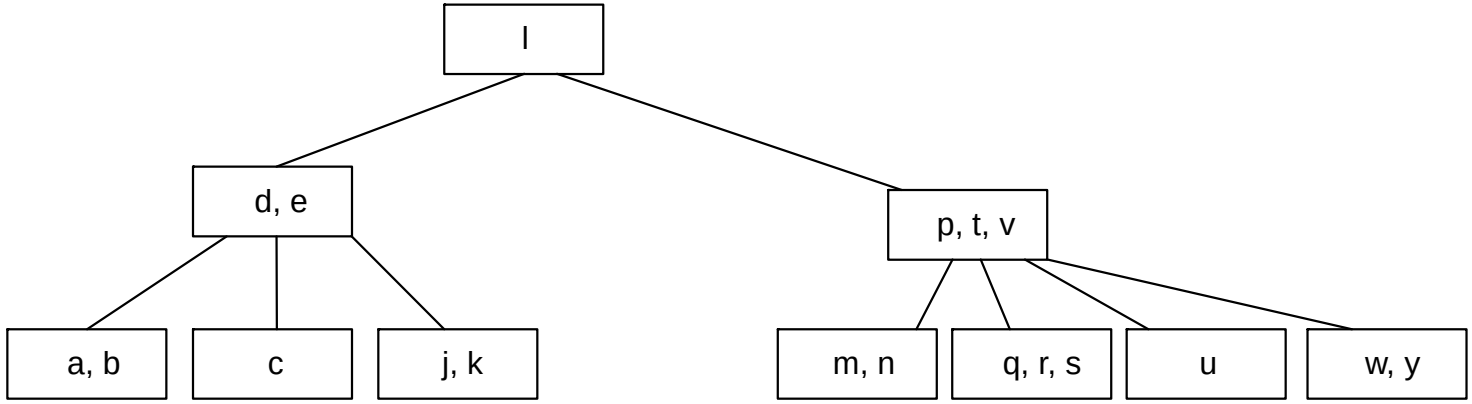
2-4 Trees

– E.g. delete “g”: remove the node and send the parent key down to a sibling node



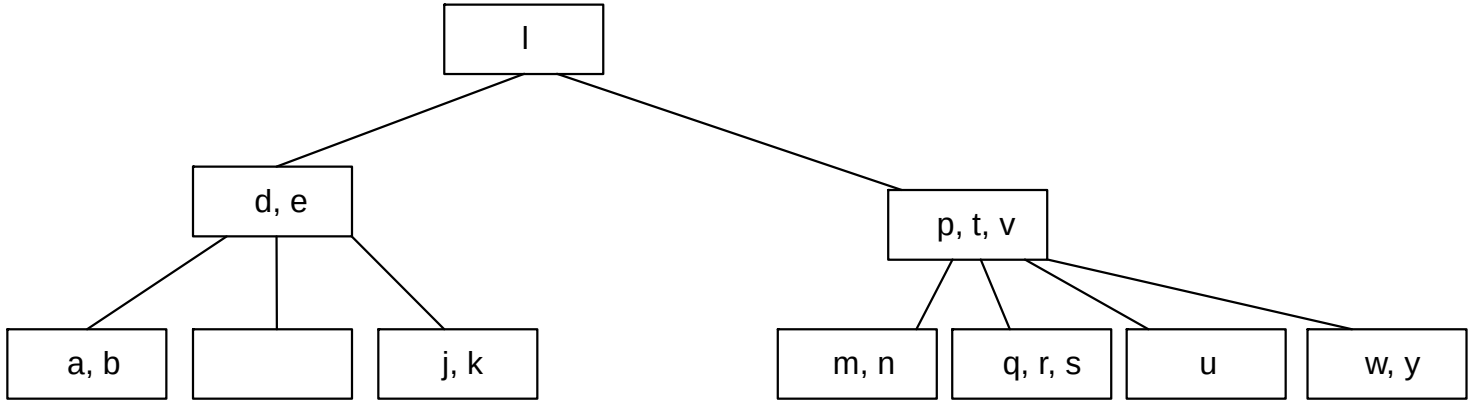
2-4 Trees

– E.g. delete “c”: internal, swap with “d”



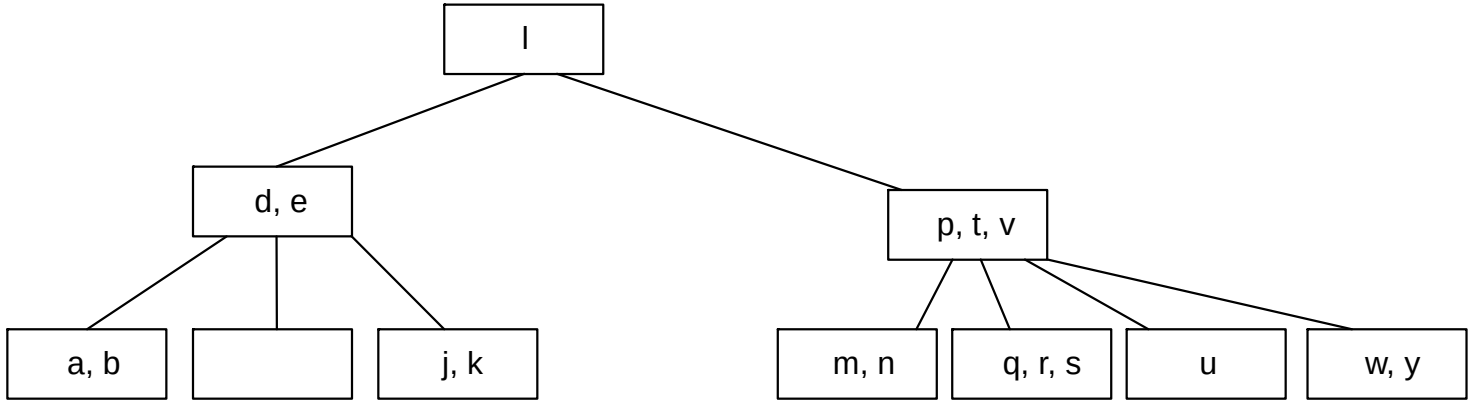
2-4 Trees

– E.g. delete “c”: 1-node



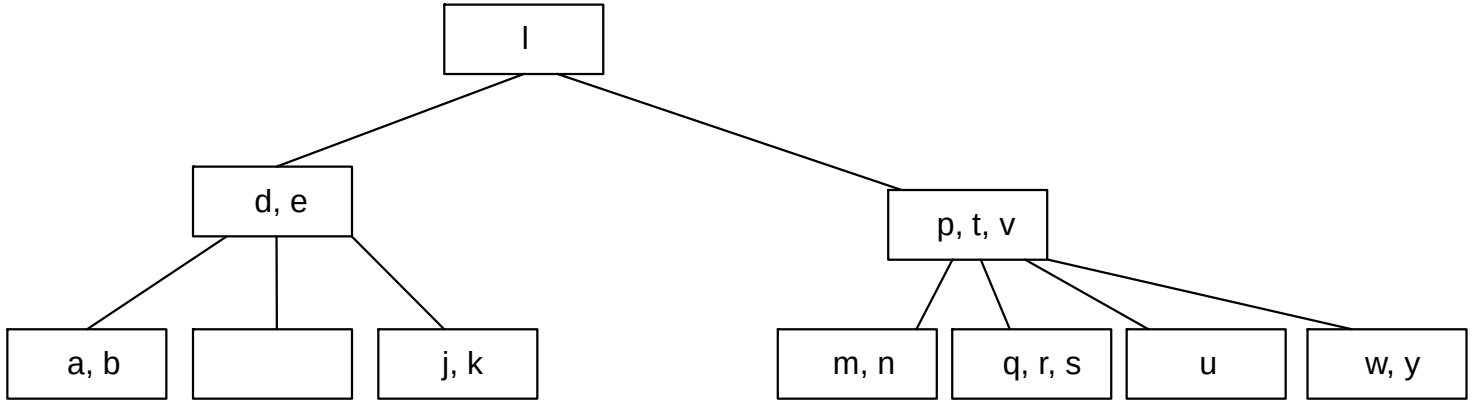
2-4 Trees

– E.g. delete “c”: 1-node



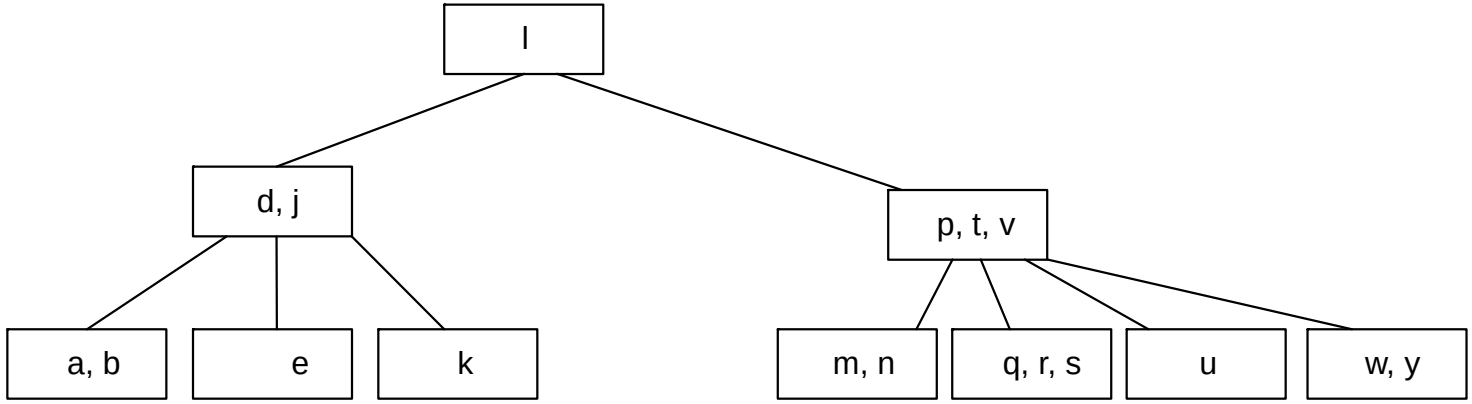
2-4 Trees

– E.g. delete “c”: sibling has more than 1 key



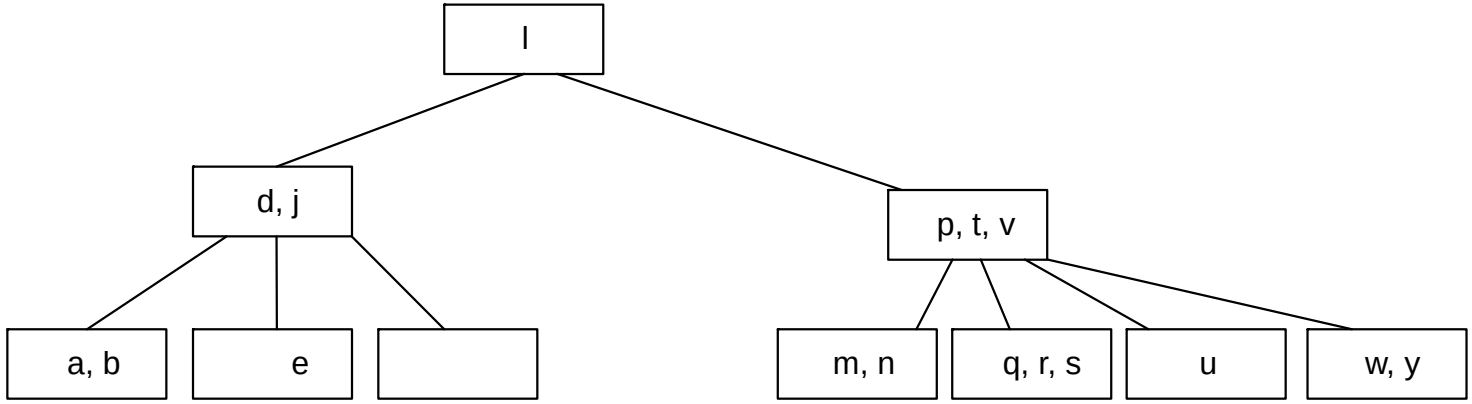
2-4 Trees

– E.g. delete “c”: send a key from sibling to parent, and key from parent to the node



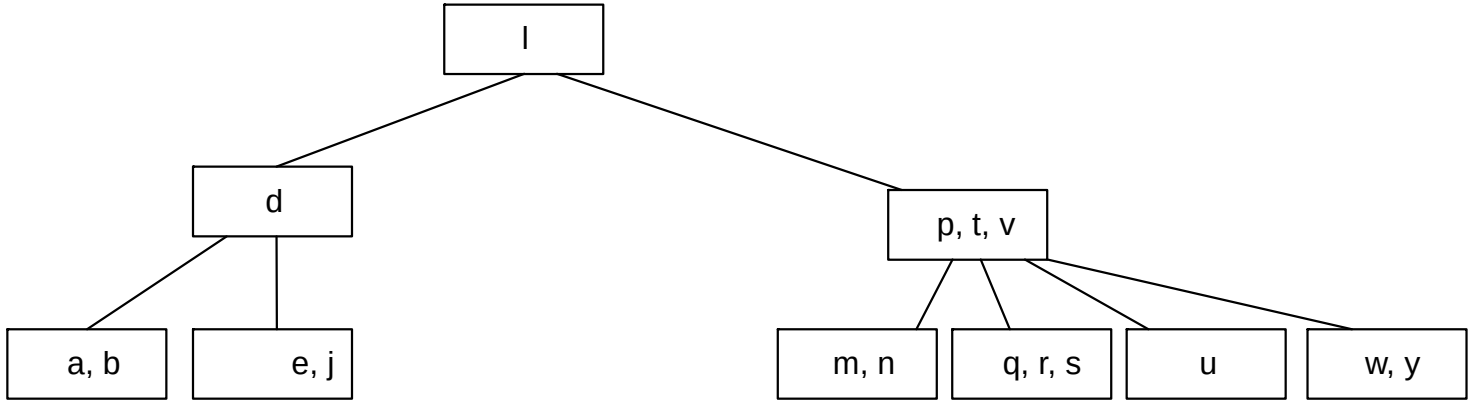
2-4 Trees

– E.g. delete “k”: send parent key down to sibling node



2-4 Trees

– E.g. delete “k”: send parent key down to sibling node



2-4 Trees: Efficiency

– Height of tree is O(log *n*)

– Searching:

- Each node checked takes O(1)
- Search takes O(log *n*)

– Insertion:

- Split takes O(1) at each level
- At most log *n* splits (1 per level)
- Insertion takes O(log *n*)

– Deletion

- Merge takes O(1)
- At most log *n* merges
- Deletion takes O(log *n*)

CSCI203

- B-Trees
  - Rudolph Bayer and Ed McGreight
- An *m*-ary search tree with the following additional properties
  - The root is either a leaf or has at least 2 children
  - All other internal nodes have at least $\lceil \frac{m}{2} \rceil$ subtrees
  - All non-root nodes have between $\lceil \frac{m}{2} \rceil$ -1 and *m-1* keys inclusive
  - All leaves are at the same level

B-Trees: Searching

- Similar to searching a binary tree
- If searching for value, compare value with each key
  - If value == key return the associated data
  - If value < key recursively search the subtree left of key
  - If value > key, repeat the process using the next key, if there is no next key, search the last subtree.

B-Trees: Insertion

- Find the leaf where the item is to be inserted
- Insert the item
- If the node overflows (now has *m* keys)
  - If an immediately adjacent sibling is not full, send a key from the parent down to this sibling, and a key up from the node to the parent. Relink if necessary.
  - If both such siblings are full, split the node into two by passing the median key up to the parent
    1. Repeat with the parent if necessary.
    2. Create a new root layer if necessary.

B-Trees: Deletion

– Find where the item to be deleted is located

– If this is an internal node, swap the item with its immediate inorder successor

– Delete the item

– If the node underflows (less than $[\frac{m}{2}]$ -1 keys)

- If the node has an immediate sibling with more than $[\frac{m}{2}]$ -1 keys, send a key from the sibling up to the parent, and a key from the parent down to the node. Relink if necessary.
- If not, merge the node and one of its immediate siblings, and send a key down from the parent, placing it in between the keys from the merged nodes.
- This may cause the parent to underflow
- Fix this recursively
- If the root becomes empty, remove it

B-Trees: Efficiency

– Height of tree is O(log *n*)

– Searching:

- Each node checked takes O(1)
- Search takes O(log *n*)

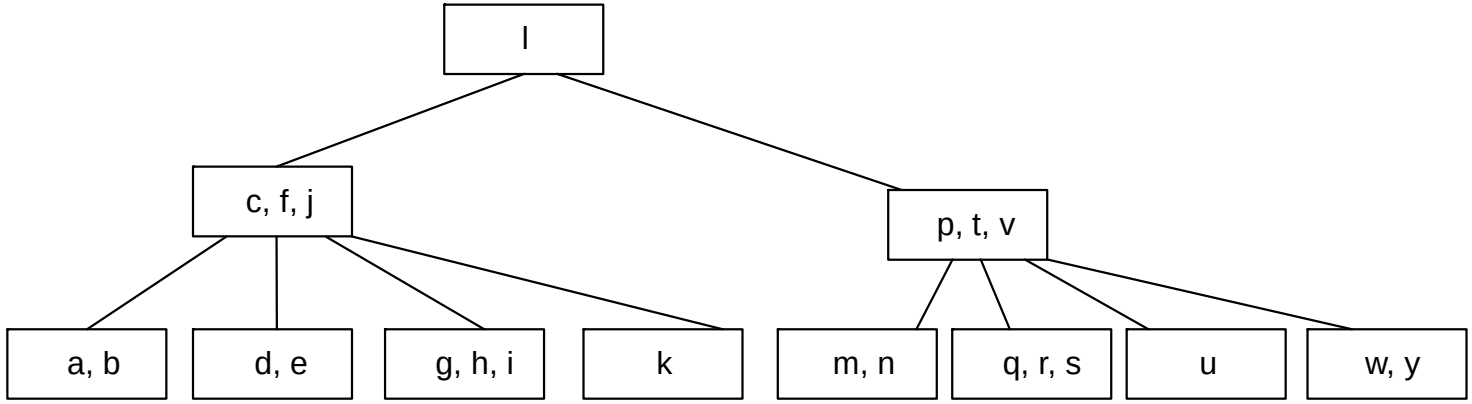
– Insertion:

- Split takes O(1) at each level
- At most log *n* splits (1 per level)
- Insertion takes O(log *n*)

– Deletion

- Merge takes O(1)
- At most log *n* merges
- Deletion takes O(log *n*)

A 2-4 Tree is an order 4 B-Tree



Heap

In computer science, a heap is a specialized tree-based data structure that satisfies the heap property: If A is a parent node of B, then the key (the value) of node A is ordered with respect to the key of node B with the same ordering applying across the heap. A heap can be classified further as either a "max heap" or a "min heap". In a max heap, the keys of parent nodes are always greater than or equal to those of the children and the highest key is in the root node. In a min heap, the keys of parent nodes are less than or equal to those of the children and the lowest key is in the root node.

Binary Heap

A binary heap is defined as a binary tree with two additional constraints:

Shape property: a binary heap is a *complete binary tree*; that is, all levels of the tree, except possibly the last one (deepest) are fully filled, and, if the last level of the tree is not complete, the nodes of that level are filled from left to right.

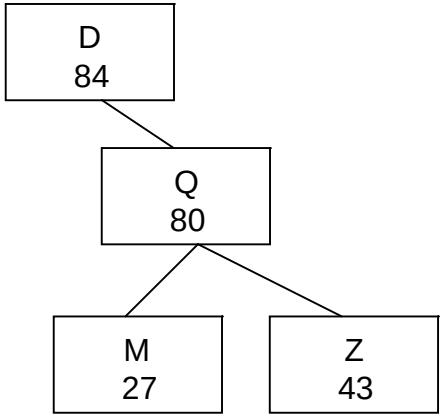
Heap property: the key stored in each node is either greater than or equal to (≥) or less than or equal to (≤) the keys in the node's children, according to some total order.

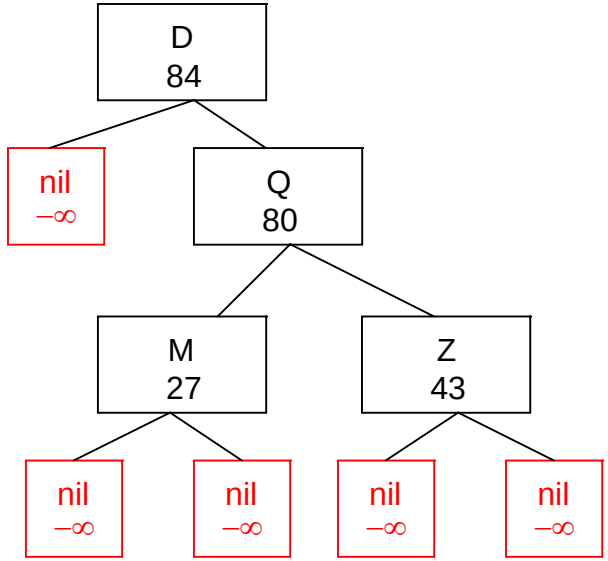
Treaps

- A hybrid data structure combining some of the properties of a binary search tree and a heap
- Each node contains a value, a left and right child pointer and a (random) priority
- AVL style single rotations are used to maintain the heap property and the binary search tree property at the same time
- Every “empty” branch has a sentinel node attached.

Treaps: sentinel nodes

This……. is really…….. this



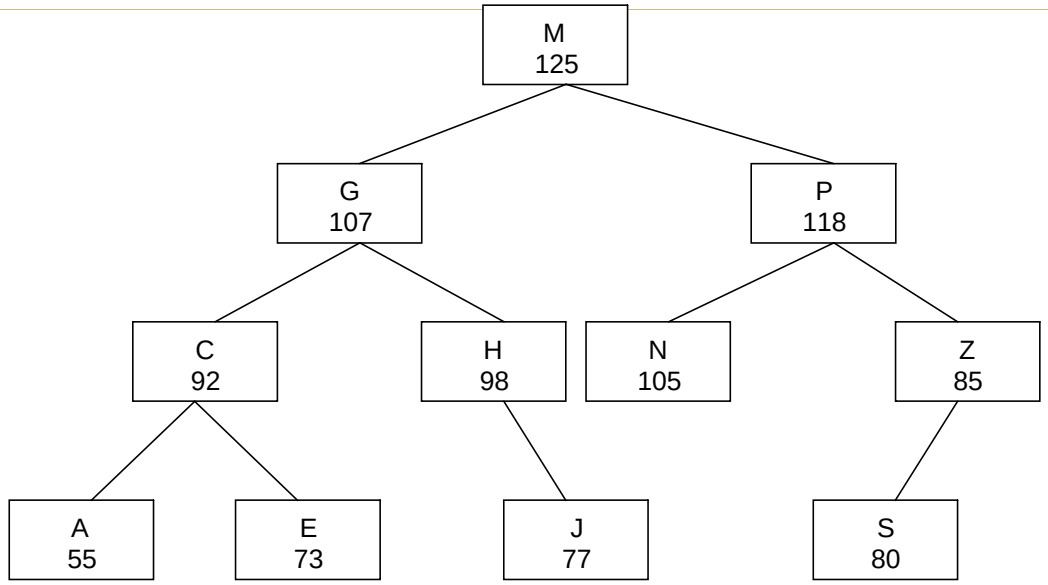


- Needed for deletions

Implementation : simple

- treap: ^treap_node
- type treap_node: record
  value: stuff
  priority: integer
  left_child, right_child: ^treap_node

Treaps: an example



– Binary search tree on letters
– Heap on priorities

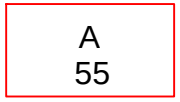
Treaps: Insertion

– function treap_insert(key, treap)

```
   if treap = nil then
   priority = random_integer()
   treap = new treap_node(key, priority,nil,nil)
else if key < treap^.value then
   treap^.left=treap_insert(key, treap^.left)
   if (treap^.left).priority > treap^.priority then
   rotate_left_child(treap)
else if treap^.value < key then
   treap^.right=treap_insert(key, treap^.right)
   if (treap^.right).priority > treap^.priority then
   rotate_right_child(treap)
else
   ; // duplicate, do nothing
   return treap;
```

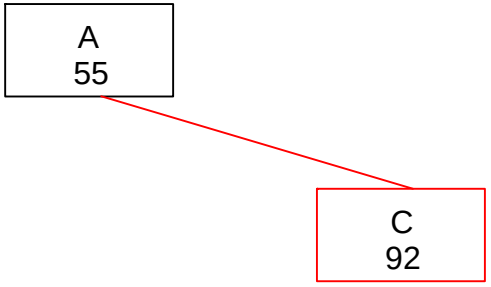
Building a treap: in alphabetical order!

– Insert “A”



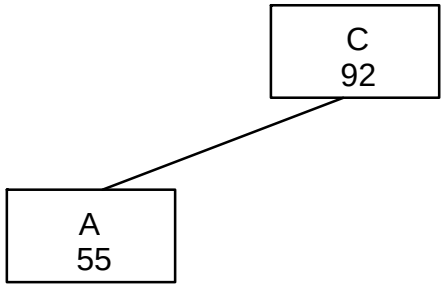
Building a treap:

- Insert “C”
- Not a heap



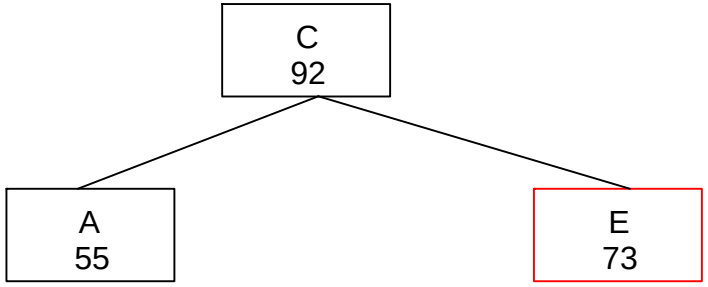
Building a treap:

– Rotate right child



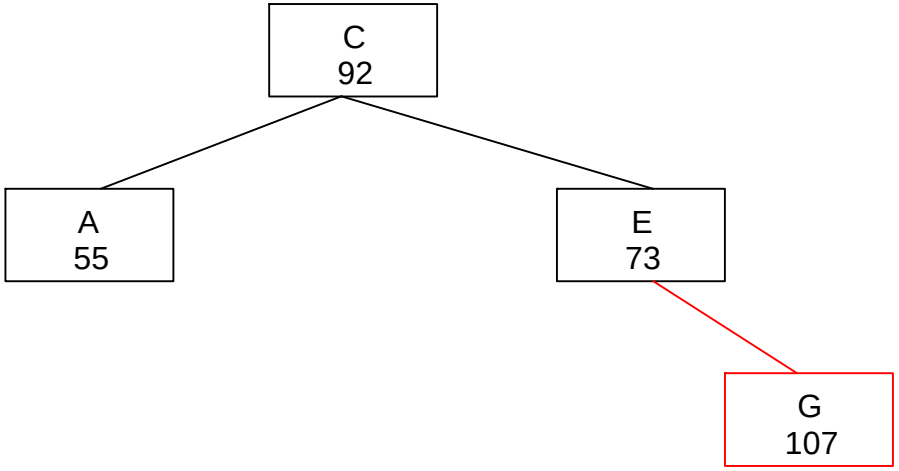
Building a treap:

– Insert “E”



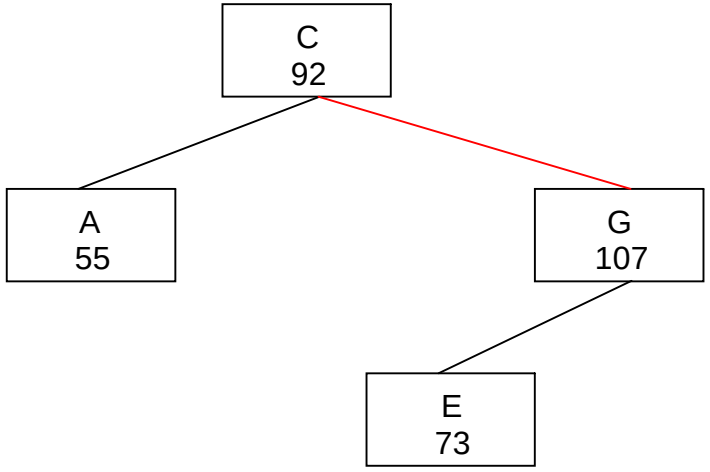
Building a treap:

- Insert “G”
- Not a heap



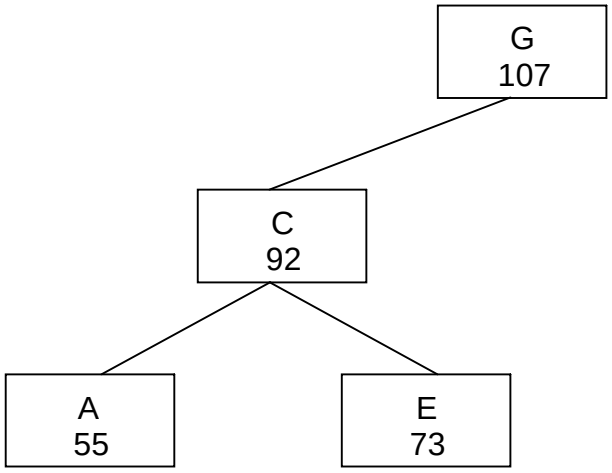
Building a treap:

– Rotate right child
– Still not a heap



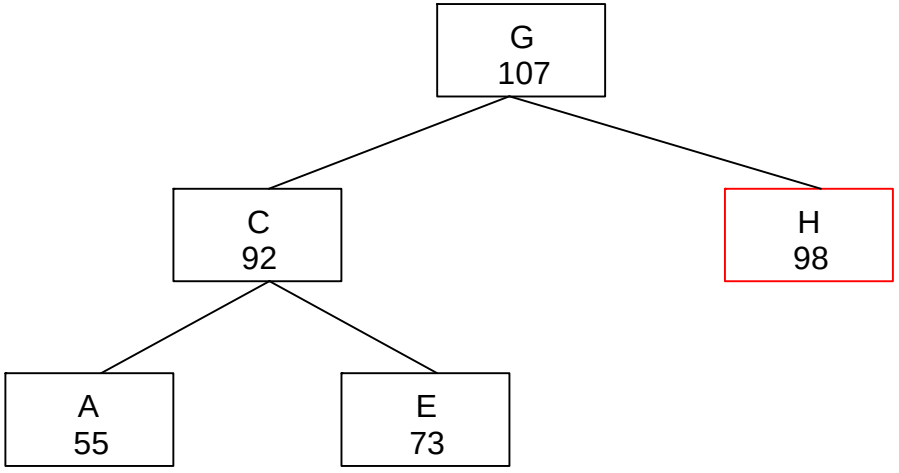
Building a treap:

– Rotate right child



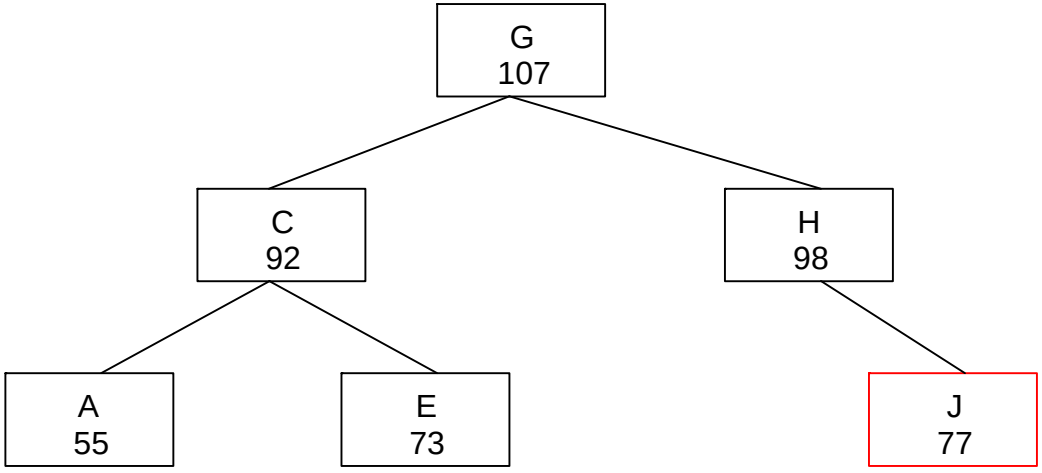
Building a treap:

– Insert “H”



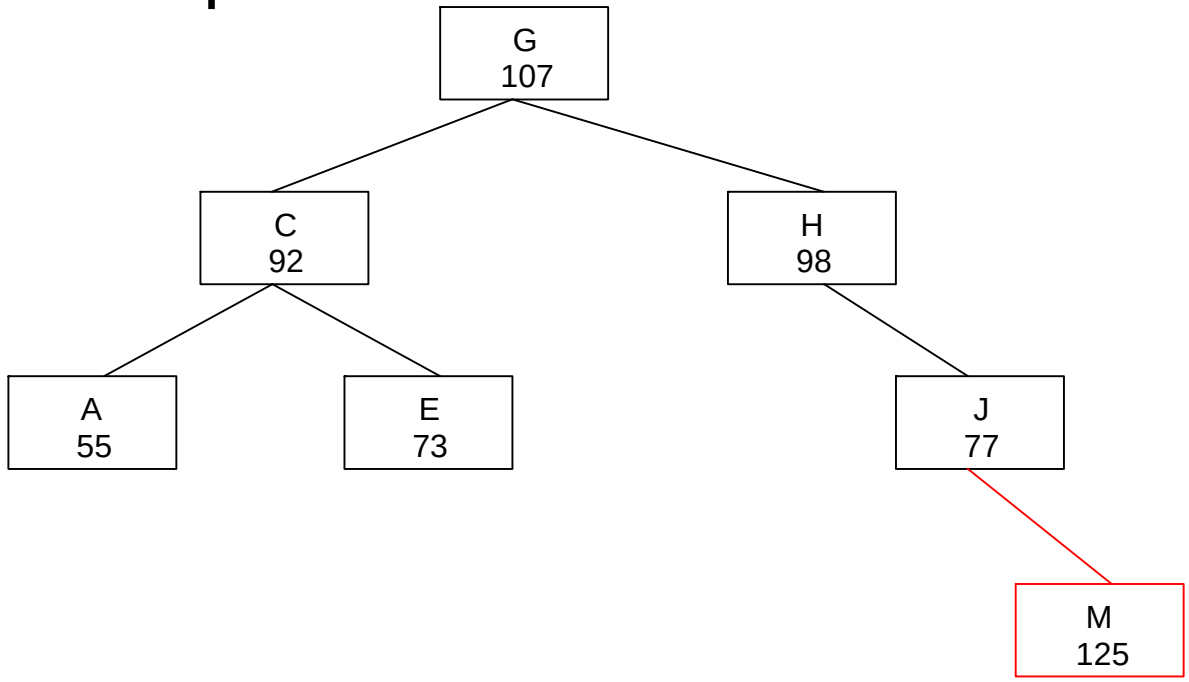
Building a treap:

– Insert “J”



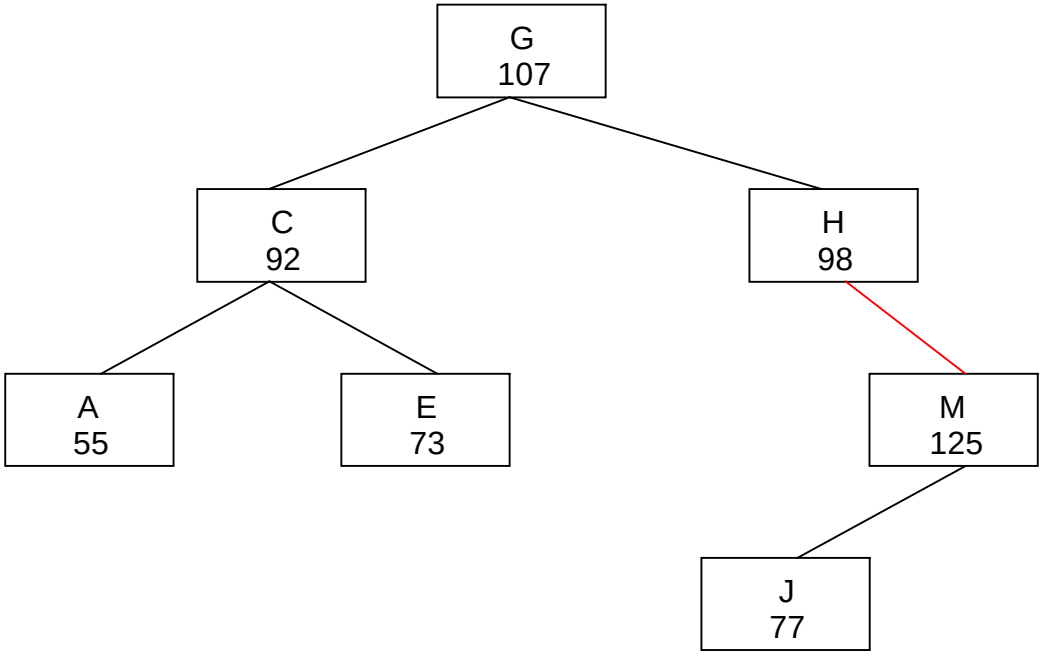
Building a treap:

– Insert “M”
– Not a heap



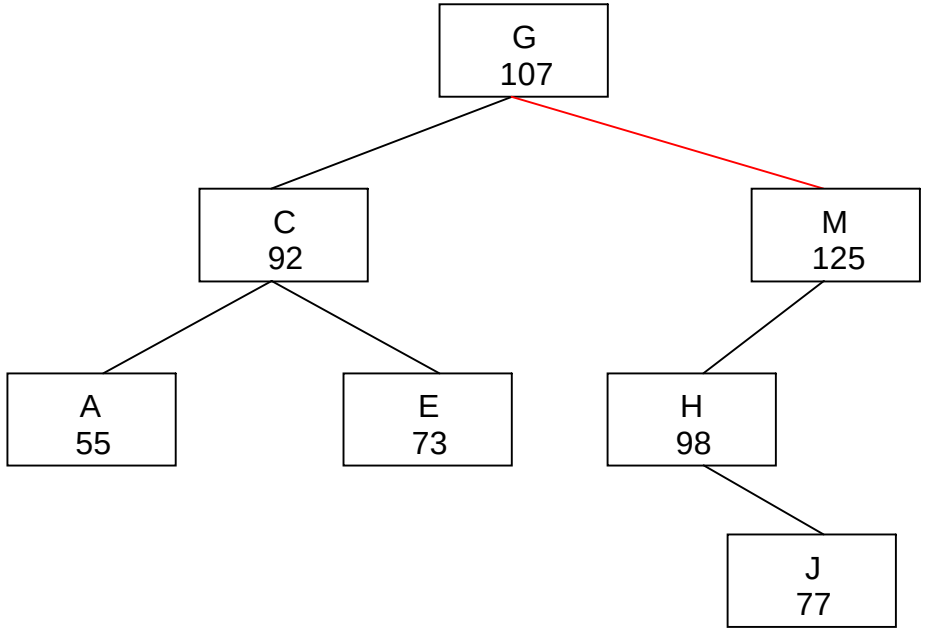
Building a treap:

– Rotate right child
– Still not a heap



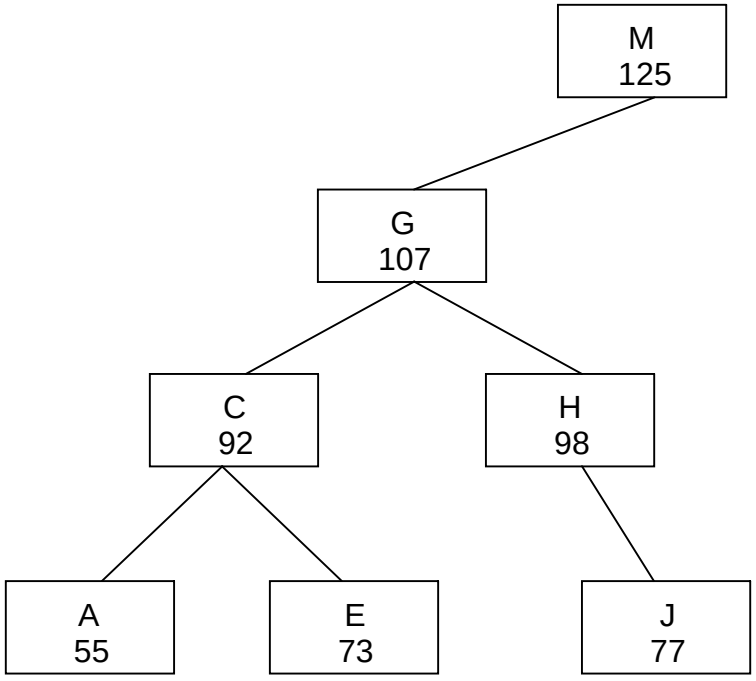
Building a treap:

– Rotate right child
– Still not a heap



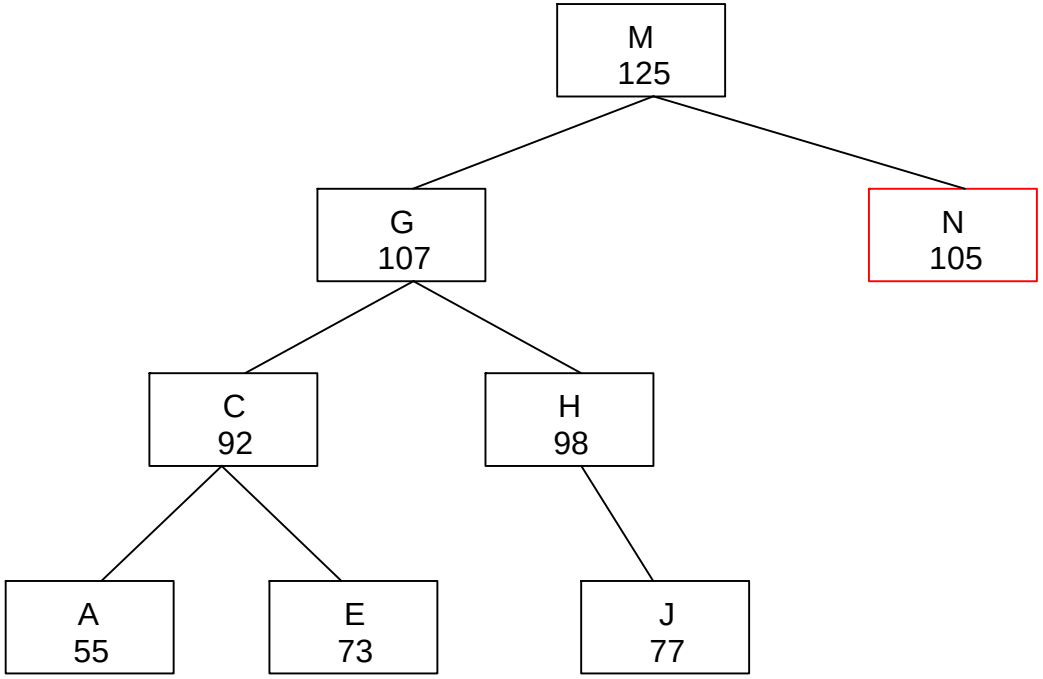
Building a treap:

– Rotate right child



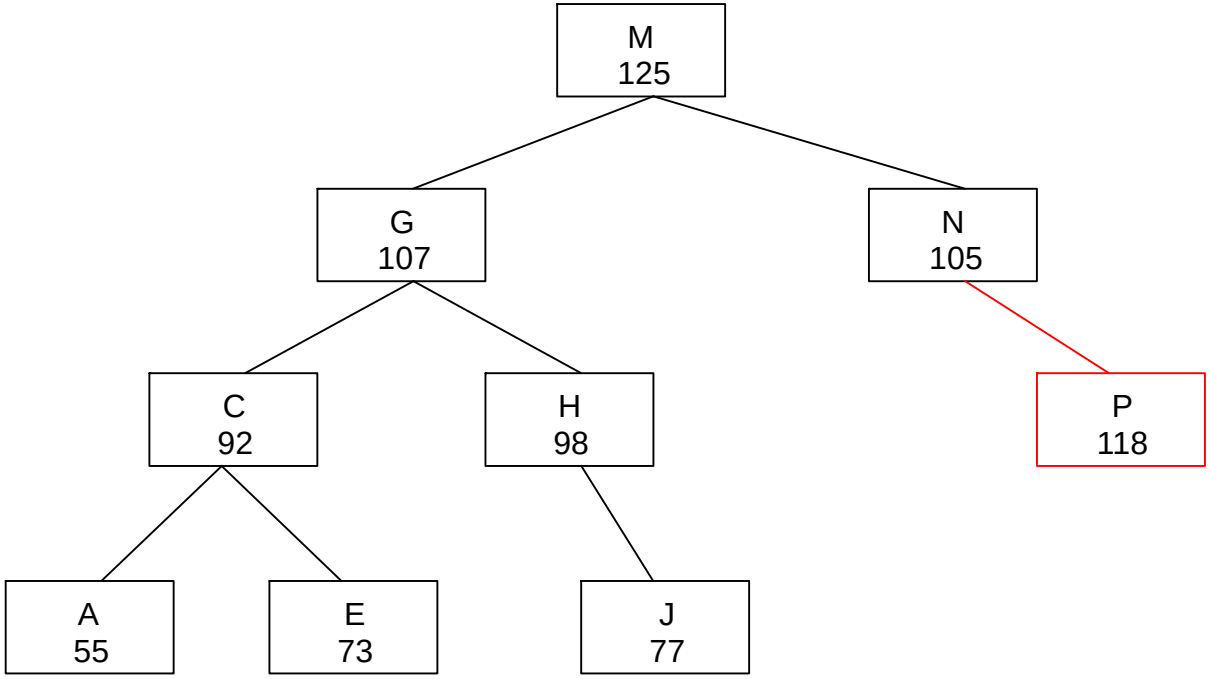
Building a treap:

– Insert “N”



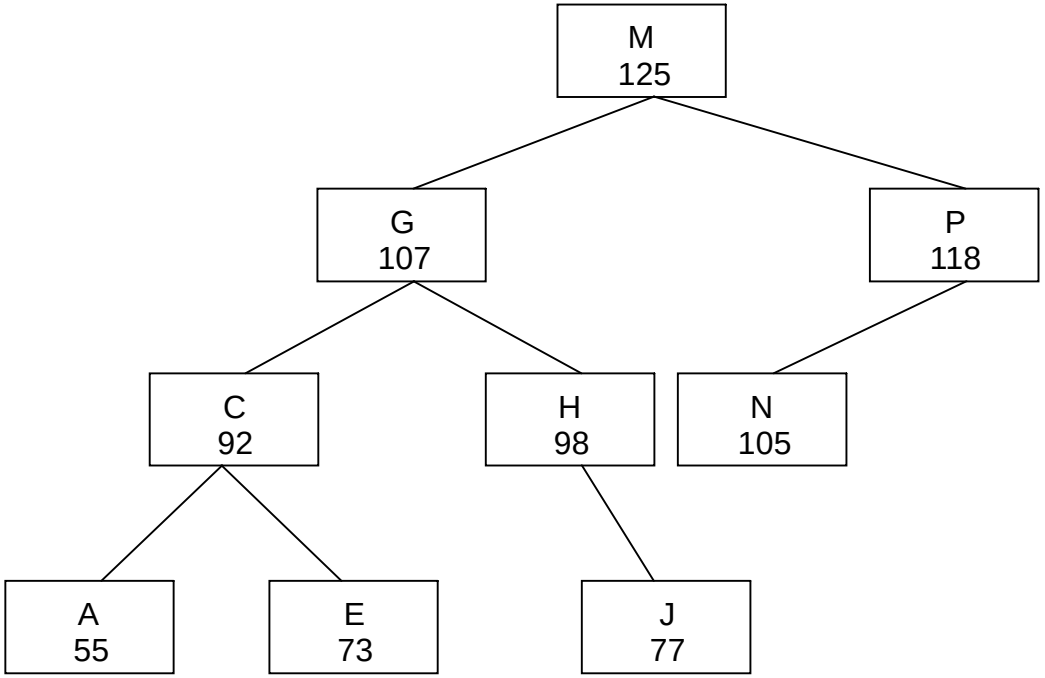
Building a treap:

– Insert “P”
– Not a heap



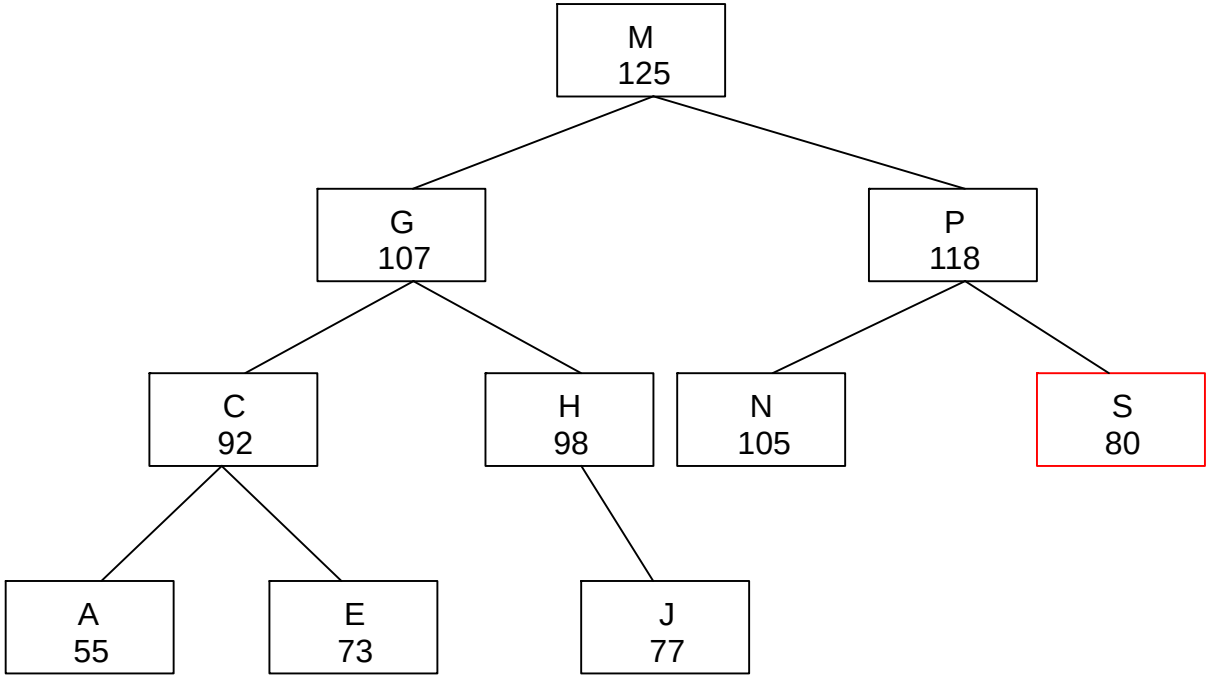
Building a treap:

– Rotate right child



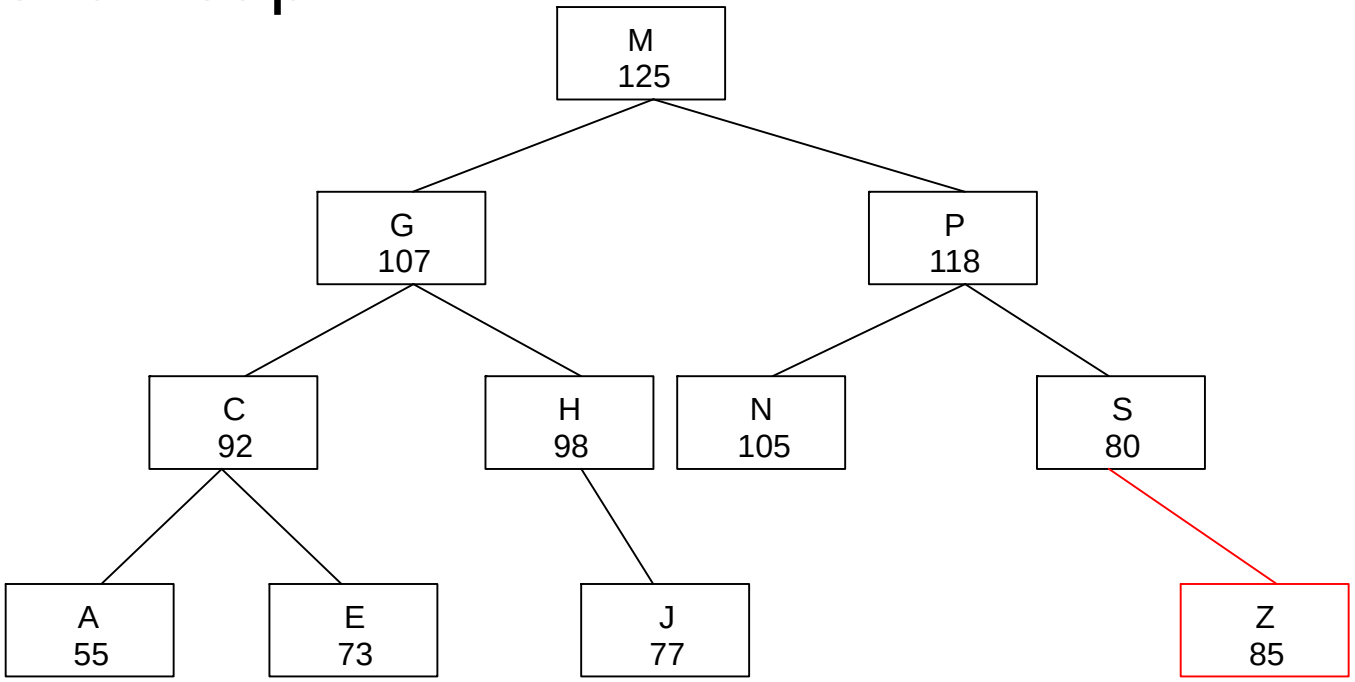
Building a treap:

– Insert “S”



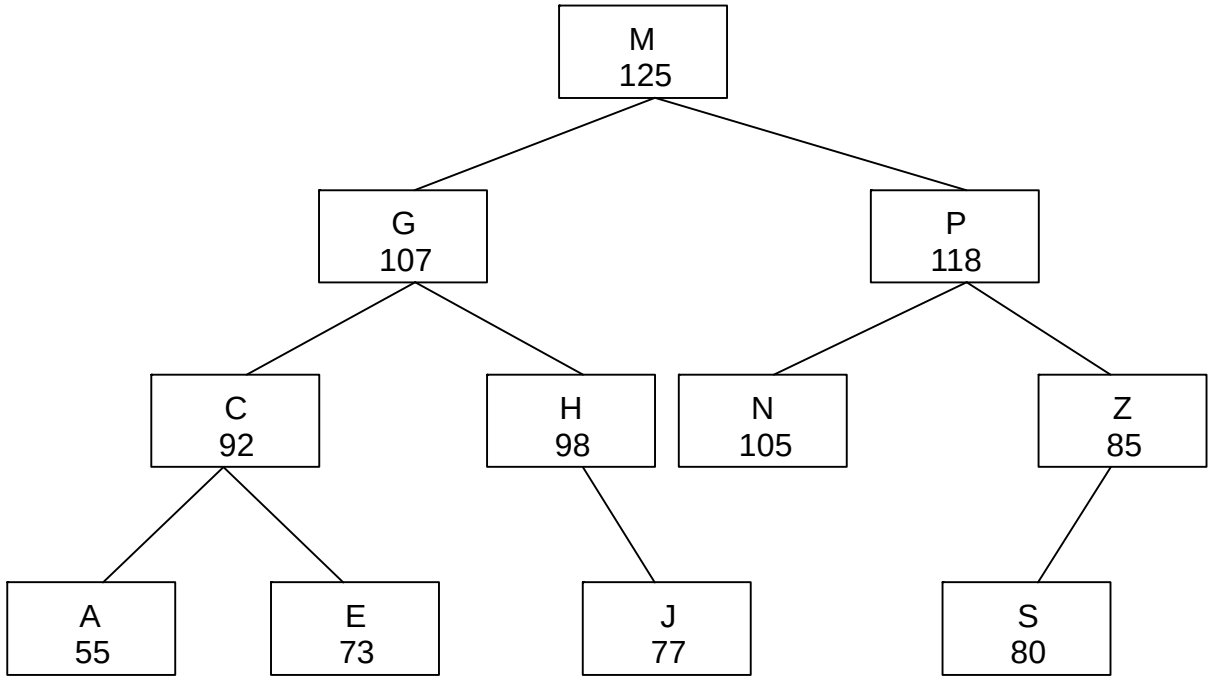
Building a treap:

– Insert “Z”
– Not a heap



Building a treap:

– Rotate right child

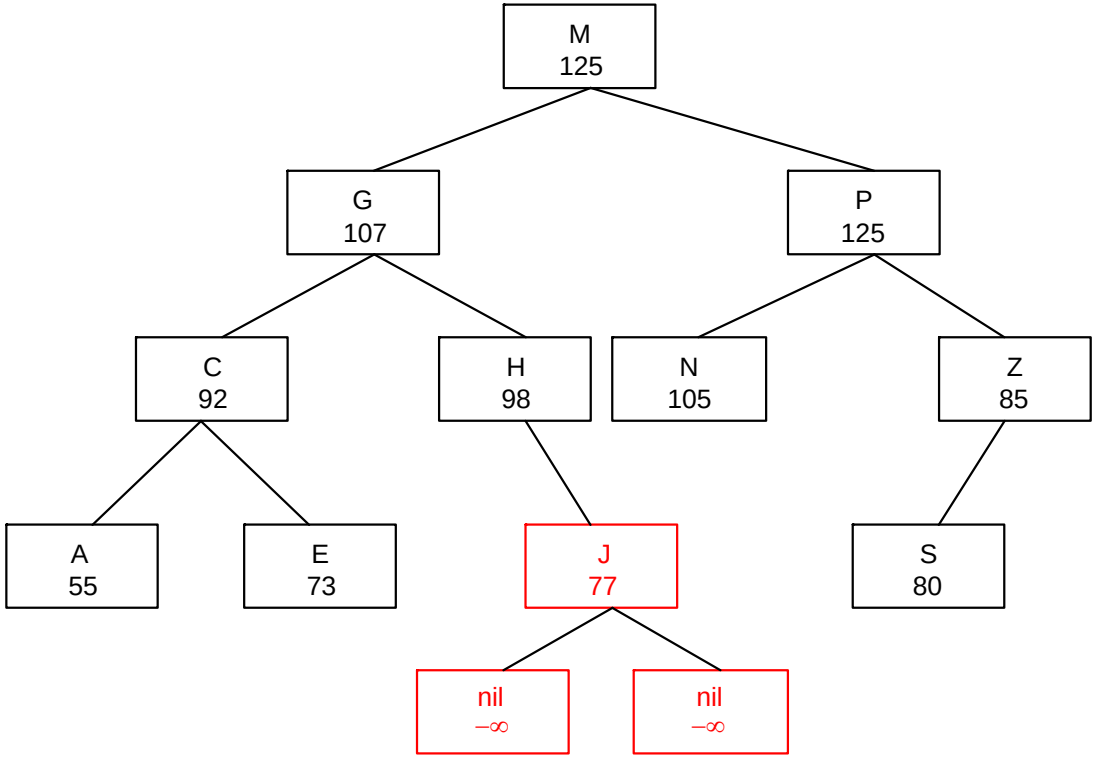


Treaps: Deletion

```
– function treap_delete(key, treap)
   if treap≠nil then
      if key < treap^.value then
         treap^.left=treap_delete(key, treap^.left)
      else if key > treap^.value then
         treap^.right=treap_delete(key, treap^.right)
      else
           if  (treap^.left)^.priority > (treap^.right)^.priority then
              rotate_left_child(treap)
           else
               rotate_right_child(treap)
           if treap≠nil then
               treap=treap_delete(key, treap)
            else
               delete treap^.left
               treap^.left = nil
     return treap
```

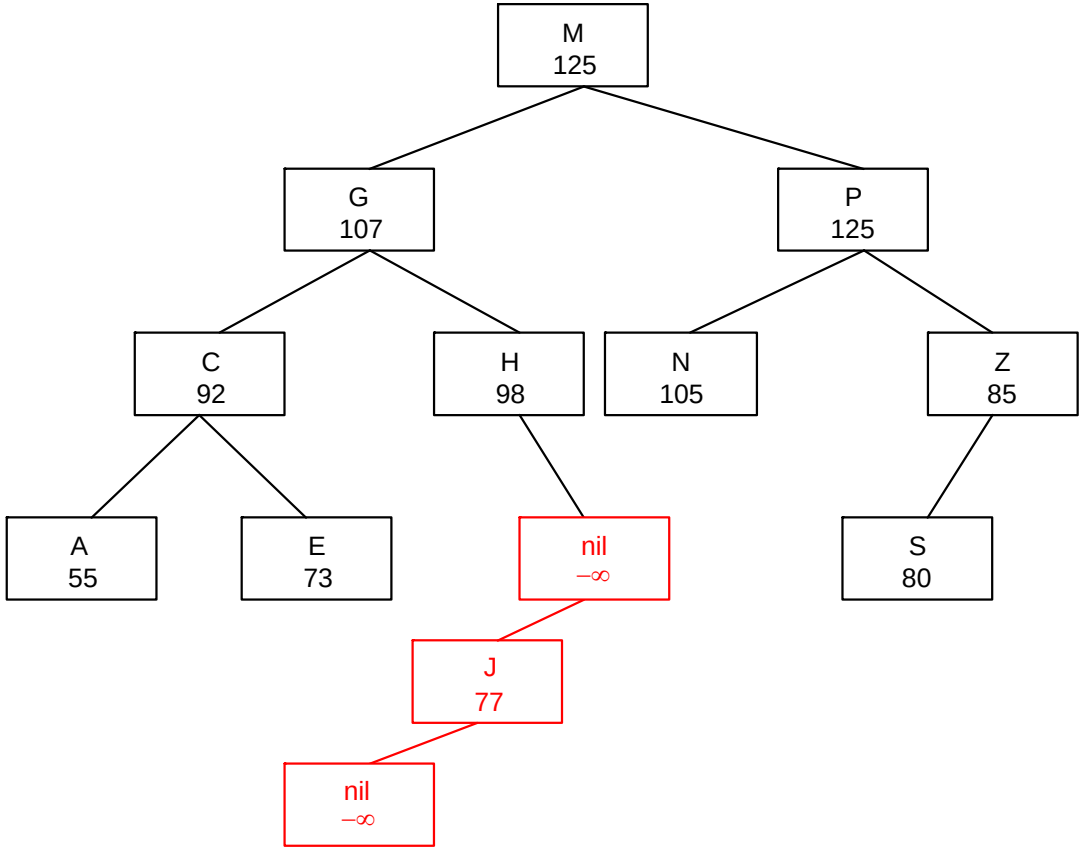
Treap deletion: an example

– Delete “J”



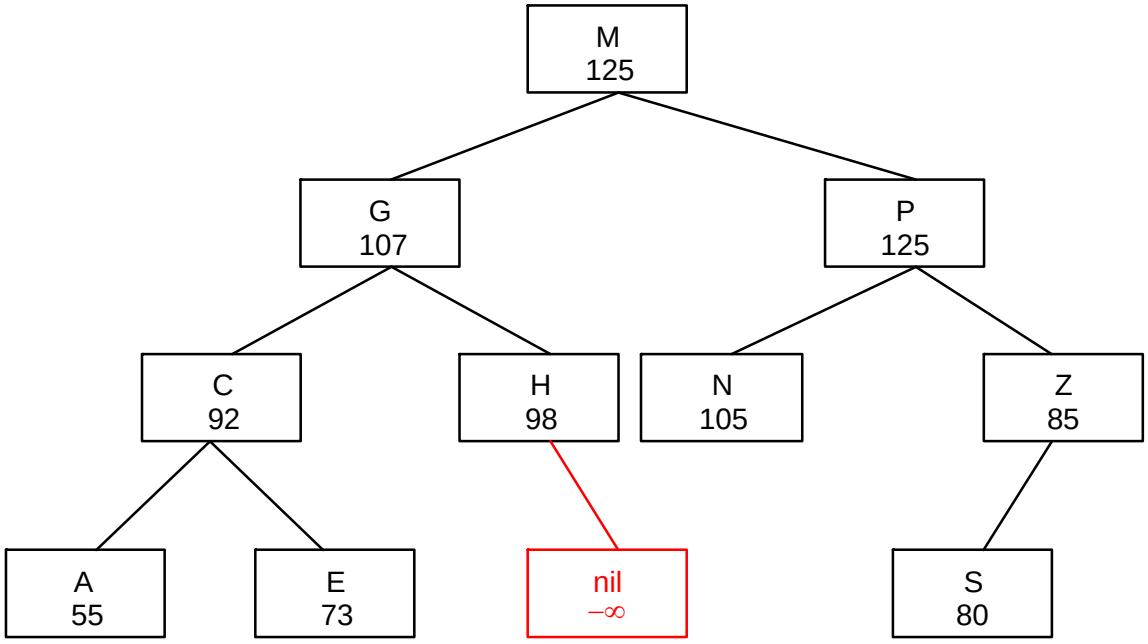
Treap deletion: an example

– Rotate right child



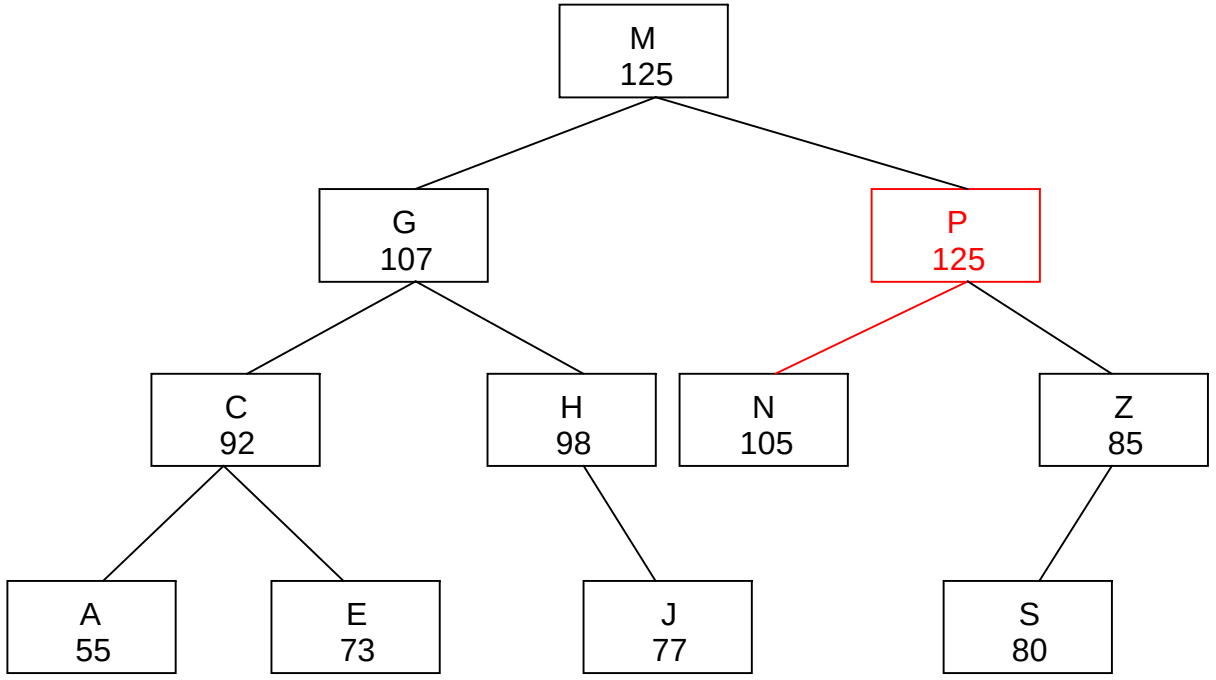
Treap deletion: an example

– Delete left child



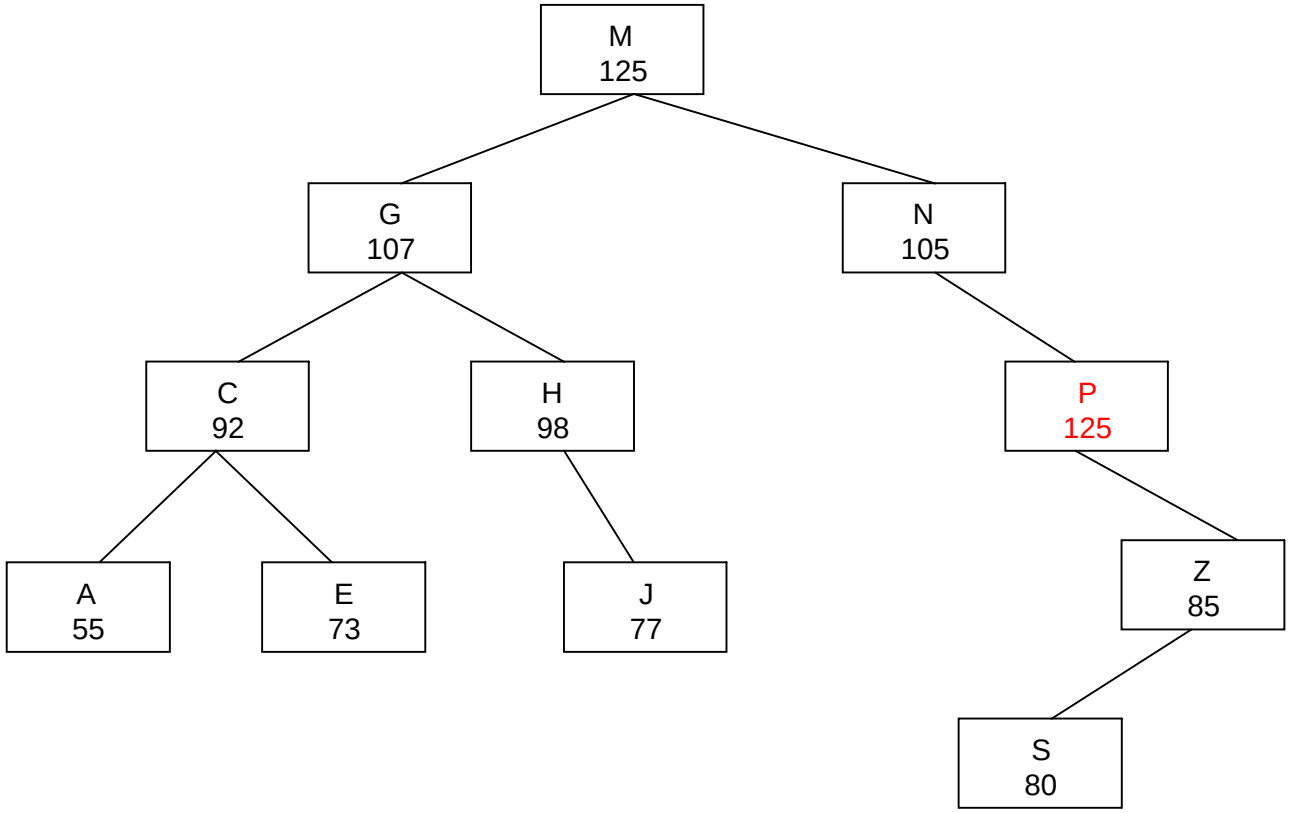
Treap deletion: an example

– Delete “P”



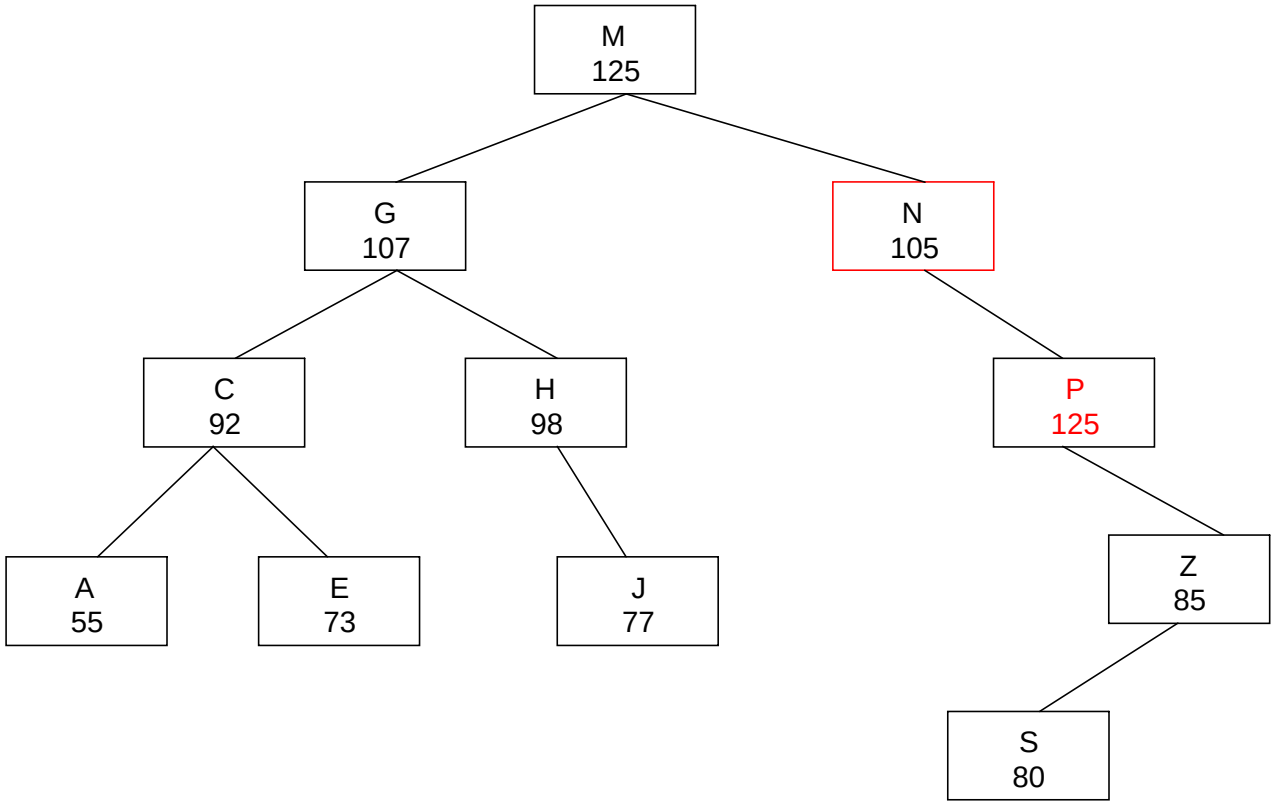
Treap deletion: an example

– Rotate left child



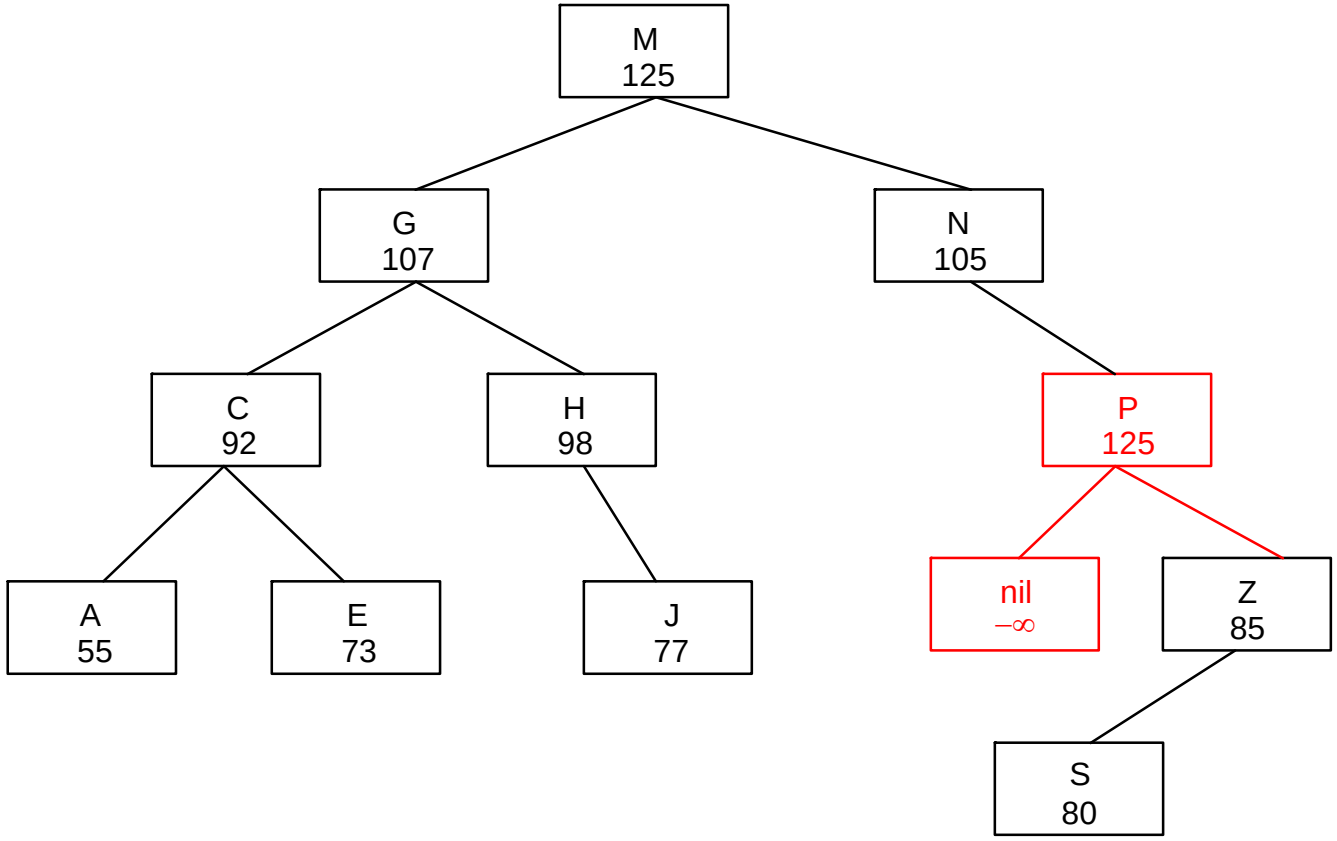
Treap deletion: an example

– Repeat from here



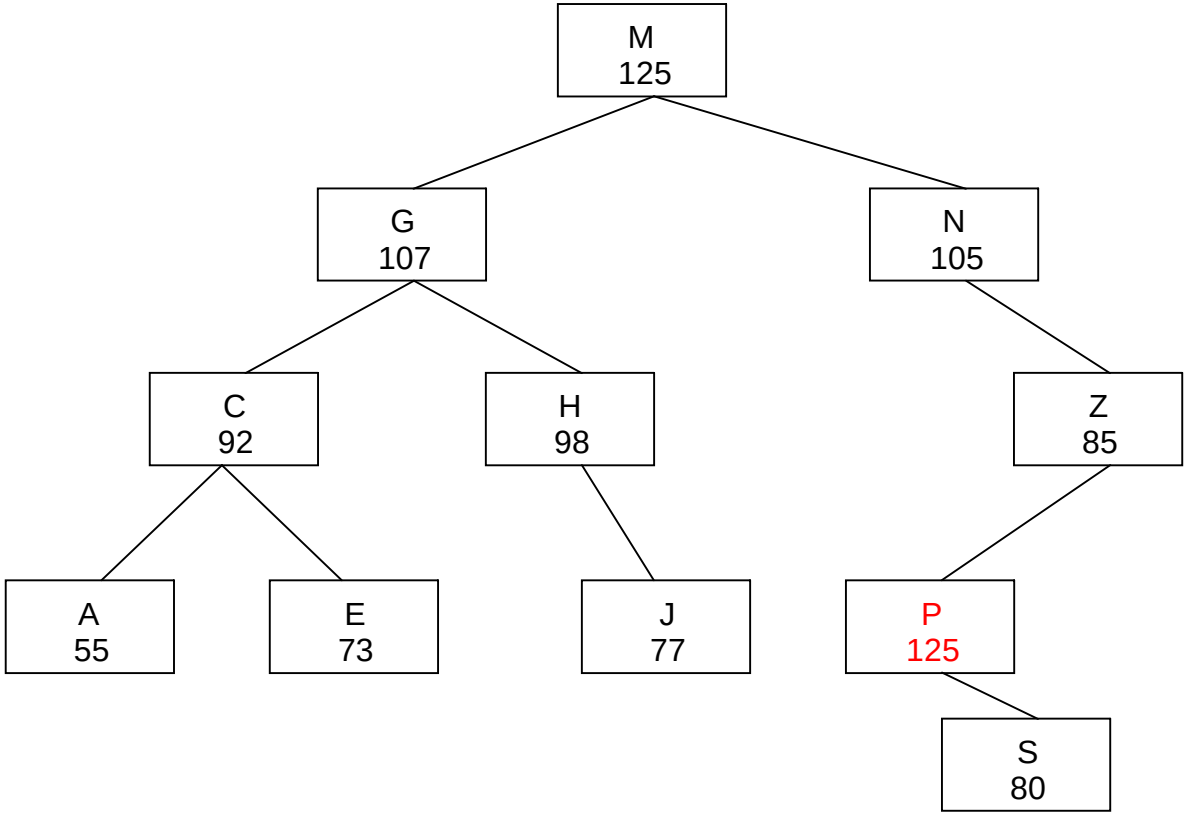
Treap deletion: an example

– Delete “P”



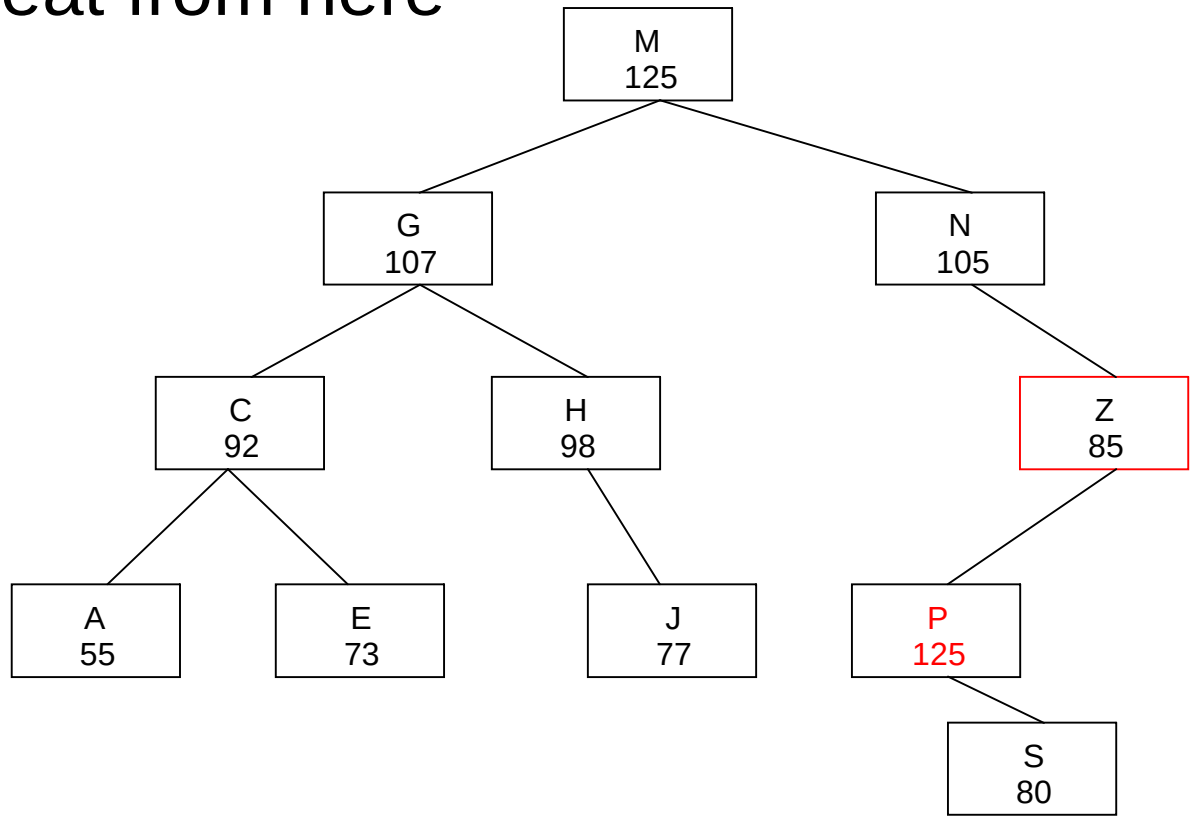
Treap deletion: an example

– Rotate right child



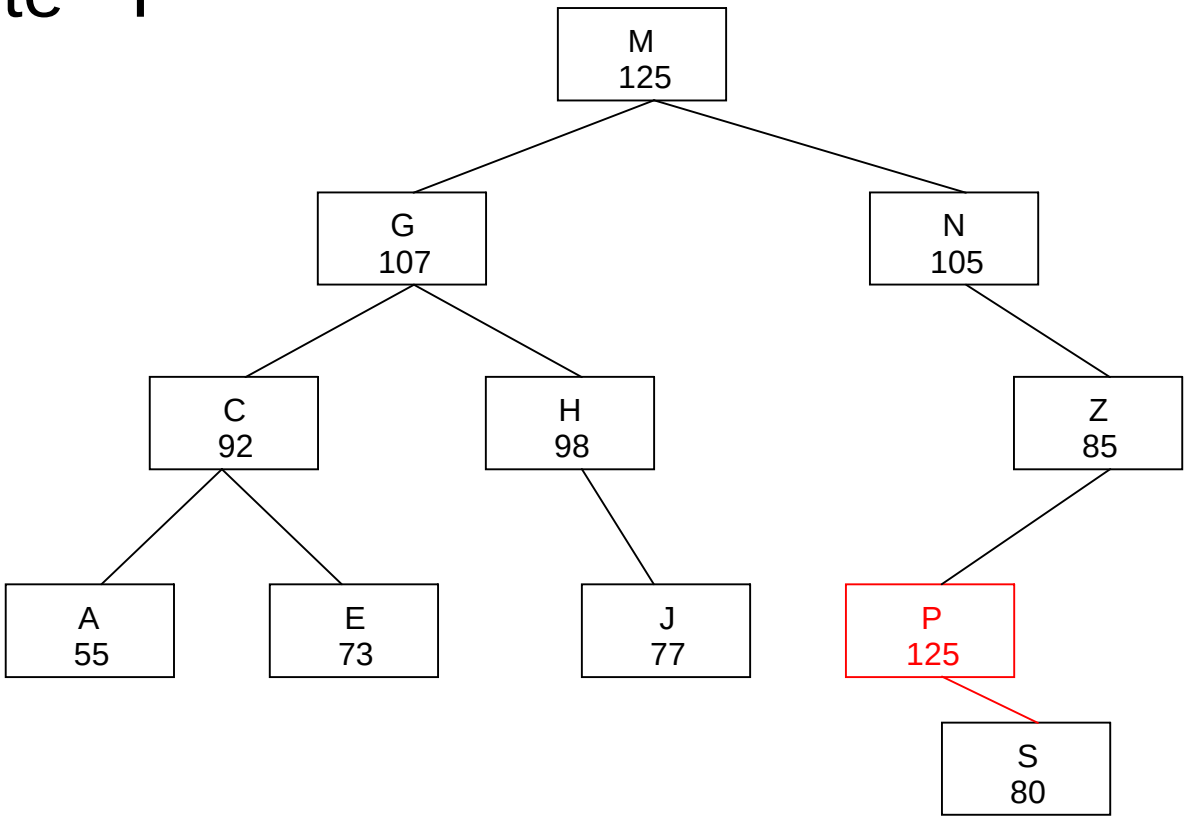
Treap deletion: an example

– Repeat from here



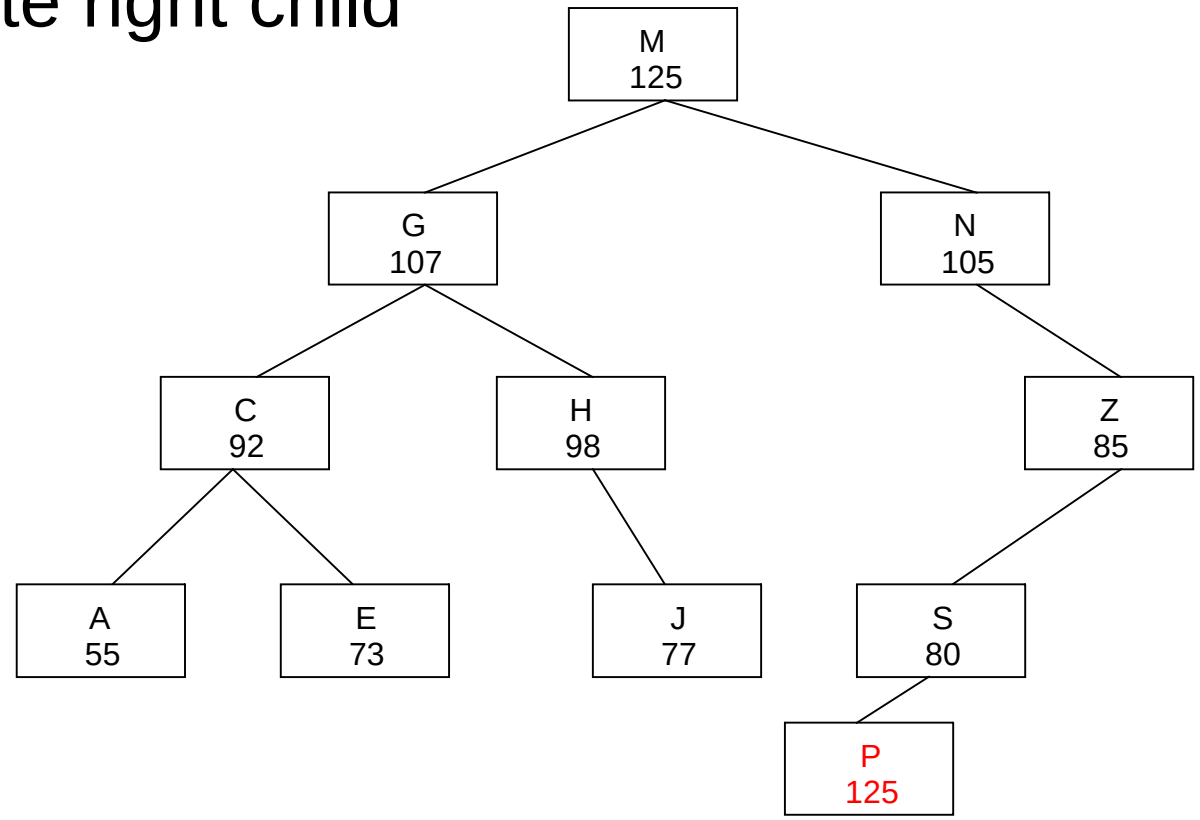
Treap deletion: an example

– Delete “P”



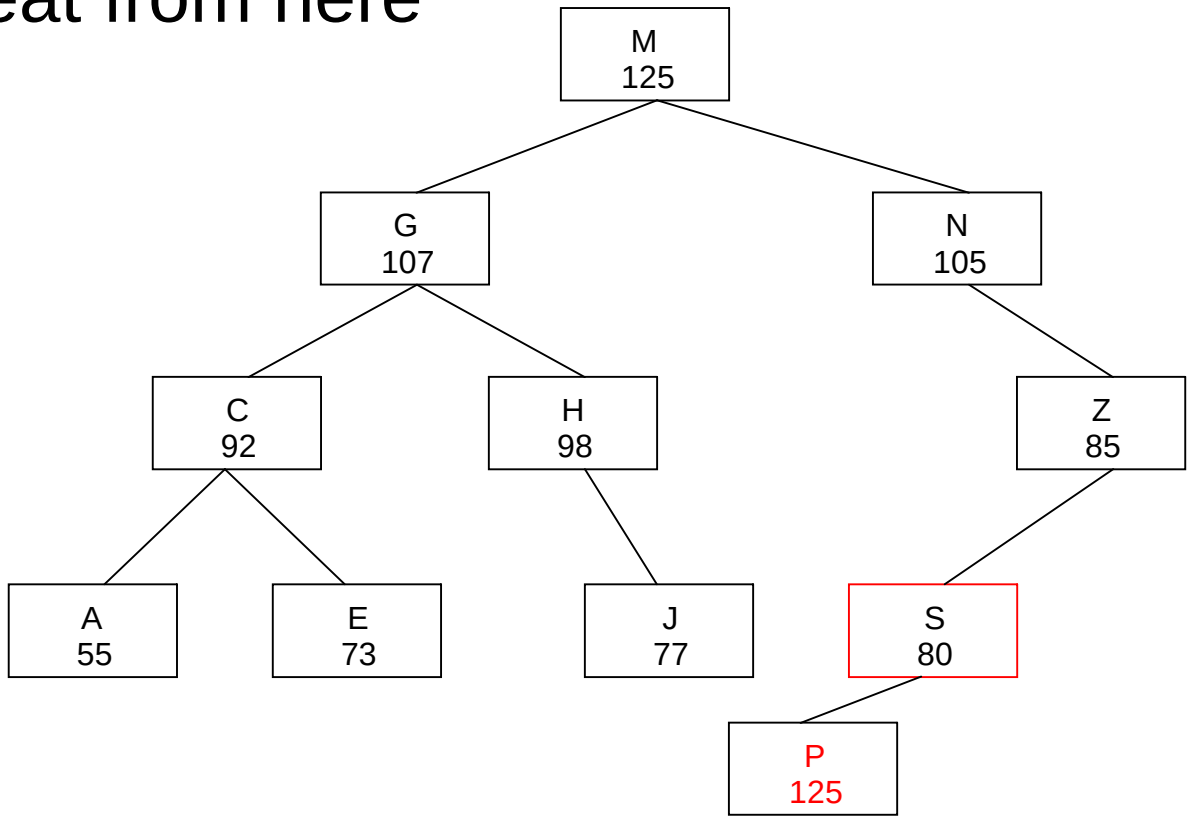
Treap deletion: an example

– Rotate right child



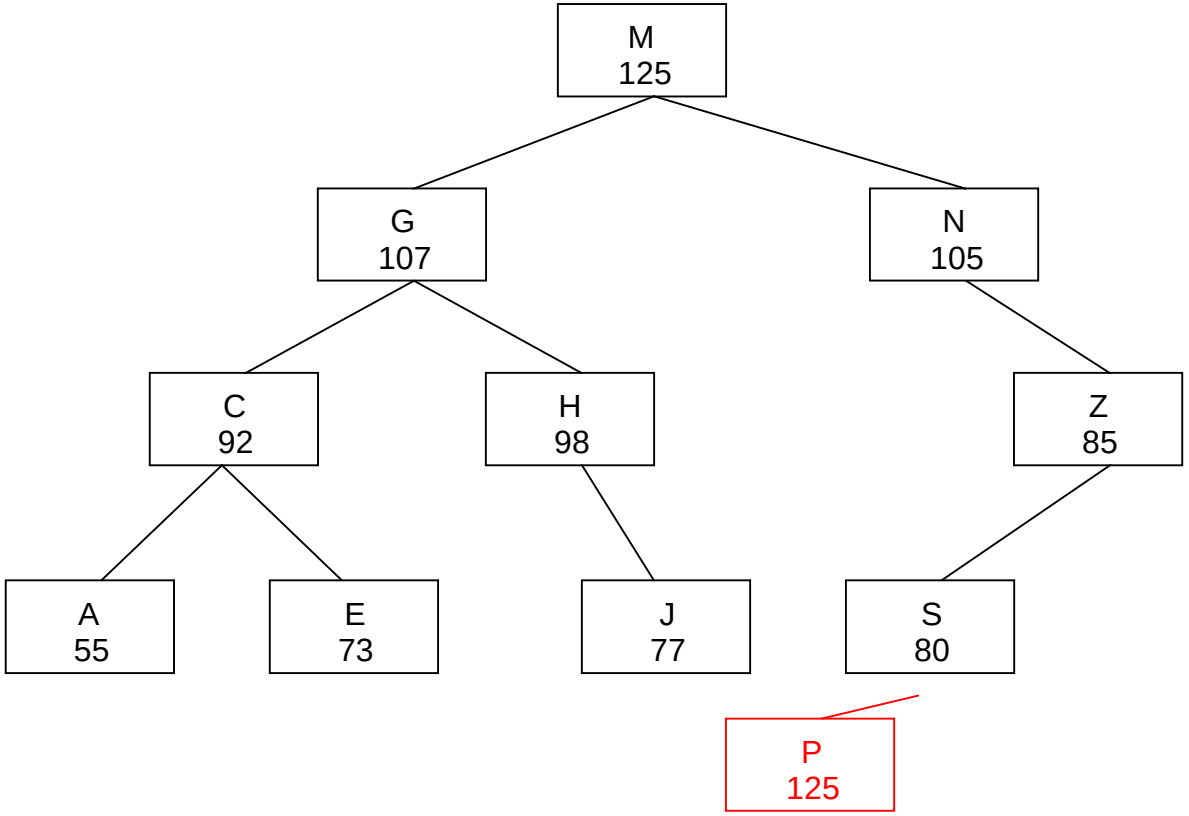
Treap deletion: an example

– Repeat from here



Treap deletion: an example

– Delete the leaf



Treaps: Efficiency

– Height of tree is O(log *n*) at best.

– Searching:

- Each node checked takes O(1)
- Search takes O(log *n*)

– Insertion:

- Search for insertion point takes O(log *n*)
- Insertion takes O(1) with possible O(1) rotation
- Insertion takes O(log *n*)

– Deletion

- Search for node takes O(log *n*)
- Rotation of node down to leaf takes O(log *n*)
- Deletion takes O(log *n*)

Homework

Assignment 2